

NAVAL POSTGRADUATE SCHOOL

Monterey, California



THESIS

A NETWORK DESIGN ARCHITECTURE FOR DISTRIBUTION OF GENERIC SCENE GRAPHS

by

Panagiotis Fiambolis
Georgios Prokopakis

September 1999

Thesis Advisor:
Co-Advisors:

Michael J. Zyda
Michael V. Capps
John S. Falby

Approved for public release; distribution is unlimited.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.

1. AGENCY USE ONLY (Leave blank)		2. REPORT DATE September 1999	3. REPORT TYPE AND DATES COVERED Master's Thesis	
4. TITLE AND SUBTITLE A NETWORK DESIGN ARCHITECTURE FOR DISTRIBUTION OF GENERIC SCENE GRAPHS			5. FUNDING NUMBERS	
6. AUTHOR(S) Fiambolis, Panagiotis and Prokopakis, Georgios				
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000			8. PERFORMING ORGANIZATION REPORT NUMBER	
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)			10. SPONSORING / MONITORING AGENCY REPORT NUMBER	
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.				
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution unlimited.			12b. DISTRIBUTION CODE	
13. ABSTRACT (maximum 200 words) <p>Sharing a common view while collaborating in networked virtual environments is complex. The SOFT project examines a new approach: using generic scene graphs as a bus, for graphics distribution. This thesis (as part of the SOFT project) examines network architecture for distribution of generic scene graphs.</p> <p>We design and implement the network architecture with a centralized Java server. This server provides scalability, persistence, reliability, and latecomer support. The server provides interoperability and can support any SSGs on any platform. The extraction of information from the network layer is implemented in two ways. In the first, we use Java's inherent serialization mechanisms; in the second, we use the Dial-a-Behavior (DaBP) protocol.</p> <p>We empirically test the server's overhead with both network mechanisms. We have concluded that using DaBP significantly reduces the server's overhead by a factor of six but only for less than 50,000 packets. Moreover, the use of DaBP provides implementation flexibility because data format can change dynamically without requiring re-compilation. Finally, DaBP, while promising, must mature and be shown to reduce overhead for large number of packets before it is ready to be incorporated into the final architecture solution for SOFT.</p>				
14. SUBJECT TERMS Networking Virtual Environments			15. NUMBER OF PAGES 165	
			16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UL	

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. Z39-18

Approved for public release; distribution is unlimited

**A NETWORK DESIGN ARCHITECTURE FOR
DISTRIBUTION OF GENERIC SCENE GRAPHS**

Panagiotis Fiambolis
Major, Greek Army
B.S., Greek Army Academy, 1985

Georgios Prokopakis
Major, Greek Army
B.S., Greek Army Academy, 1982

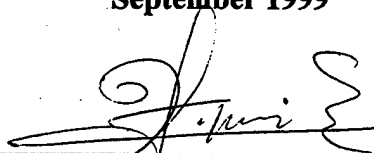
Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

**NAVAL POSTGRADUATE SCHOOL
September 1999**

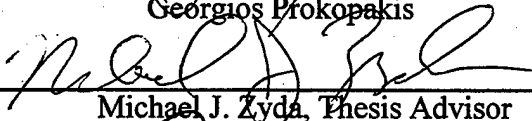
Authors:



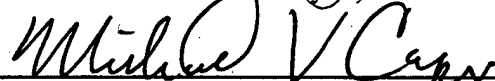
Panagiotis Fiambolis


Georgios Prokopakis

Approved by:



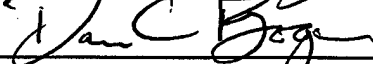
Michael J. Zyda, Thesis Advisor



Michael V. Capps, Co Advisor



John S. Falby, Co Advisor



Dan C. Boger, Chairman
Department of Computer Science

ABSTRACT

Sharing a common view while collaborating in networked virtual environments is complex. The SOFT project examines a new approach: using generic scene graphs as a bus, for graphics distribution. This thesis (as part of the SOFT project) examines network architecture for distribution of generic scene graphs.

We design and implement the network architecture with a centralized Java server. This server provides scalability, persistence, reliability, and latecomer support. The server provides interoperability and can support any SSGs on any platform. The extraction of information from the network layer is implemented in two ways. In the first, we use Java's inherent serialization mechanisms; in the second, we use the Dial-a-Behavior (DaBP) protocol.

We empirically test the server's overhead with both network mechanisms. We have concluded that using DaBP significantly reduces the server's overhead by a factor of six but only for less than 50,000 packets. Moreover, the use of DaBP provides implementation flexibility because data format can change dynamically without requiring re-compilation. Finally, DaBP, while promising, must mature and be shown to reduce overhead for large number of packets

before it is ready to be incorporated into the final architecture solution for SOFT.

TABLE OF CONTENTS

I. INTRODUCTION	1
A. OVERALL GOALS	1
1. <i>SOFT</i>	1
2. <i>NTII</i>	1
3. <i>Internet2</i>	2
B. SCOPE OF THIS THESIS	2
1. <i>Server Design</i>	2
2. <i>Multi-Tier</i>	5
C. CONTRIBUTIONS AND GOAL OF THIS THESIS	7
D. CHAPTER SUMMARY	8
II. SOFT PROJECT	9
A. OBJECTIVES	9
1. <i>Network Collaboration</i>	9
B. PROPOSED CLIENTS OF SOFT	13
1. <i>Telecubicle</i>	13
2. <i>Telesurgery</i>	18
3. <i>Generic Sharing of Stand-Alone Applications</i>	19
C. ARCHITECTURE	21
1. <i>SSG and Reporting</i>	22
2. <i>Mapping Layer (ML)</i>	22
3. <i>Network Scene Graph Layer (NSG)</i>	23
4. <i>Dial-a-Behavior Protocol (DaBP)</i>	25
III. RELATED WORK	27
A. INTRODUCTION	27
B. INTERCONNECTING PROPRIETARY STANDALONE APPLICATIONS	27
C. INTERCONNECTING HETEROGENEOUS VIRTUAL ENVIRONMENTS	30
D. BUILDING FRAMEWORKS FOR DISTRIBUTED VIRTUAL ENVIRONMENTS BASED ON A CLIENT/SERVER APPROACH AND REPLICATION	32
1. <i>Community Place (CP) Architecture</i>	32
2. <i>Spline Architecture</i>	35
3. <i>Repo-3D Architecture</i>	36
E. CONCLUSIONS	38
IV. SOFT NETWORK ARCHITECTURE	41
A. INTRODUCTION	41
B. SERVER	42
1. <i>Justification for Centralized Server</i>	42
2. <i>Justification for Java Server</i>	46
C. CLIENT SIDE	50
1. <i>API to Network</i>	50

2. Client High-Level Architecture	51
D. MULTICASTING	51
V. SERVER DESIGN AND IMPLEMENTATION.....	55
A. INTRODUCTION	55
B. METHODOLOGY	55
C. PHASES OF DEVELOPMENT	59
1. Phase One	59
2. Phase Two	62
3. Phase Three	65
4. Phase Four	69
5. Phase Five	73
VI. CLIENT SIDE ARCHITECTURE AND DESIGN.....	77
A. NETWORK SCENE GRAPH (NSG) PROTOCOL	77
1. UID	77
2. NSGtag	77
3. Owner	79
4. Contention Flags	79
5. Fields	79
B. NSG SERIALIZATION/DESERIALIZATION	80
VII. EMPIRICAL TESTING	83
A. INTRODUCTION	83
B. TESTING SERVER OVERHEAD WITH DaBP	83
C. TESTING SERVER OVERHEAD WITHOUT DaBP	84
D. PROBLEMS TESTING WITH DaBP	85
E. CONCLUSIONS	86
VIII. CONCLUSIONS AND FUTURE WORK	89
A. INTRODUCTION	89
B. CONCLUSIONS	89
C. FUTURE RESEARCH IDEAS	90
D. SUMMARY	91
APPENDIX A - JAVA CODE FOR "PHASE ONE"	93
APPENDIX B - JAVA CODE FOR "PHASE TWO"	99
APPENDIX C - JAVA CODE FOR "PHASE THREE"	103
APPENDIX D - JAVA CODE FOR "PHASE FOUR"	107
APPENDIX E - XML FILE	117
APPENDIX F - DABP CLIENTS	119

APPENDIX G - JAVA CODE FOR "PHASE FIVE"	125
APPENDIX H - JAVA CODE FOR TESTING CLIENTS	133
LIST OF REFERENCES	137
BIBLIOGRAPHY	141
INITIAL DISTRIBUTION LIST	145

LIST OF FIGURES

Figure 1:	Multithreaded Process Model	4
Figure 2:	Multi-Tier Architecture	6
Figure 3:	Common Visual Environment	10
Figure 4:	A Sample Scene of Paris and Its Corresponding Scene Graph	10
Figure 5:	SOFT Layering	12
Figure 6:	Components of a Telecubicle	16
Figure 7:	Intuitive Human Communication	16
Figure 8:	The First Four Telecubicles	17
Figure 9:	CAVE	18
Figure 10:	ImmersaDesk	18
Figure 11:	Telesurgery System Components	20
Figure 12:	Telesurgery Workstation	20
Figure 13:	Master Tool Handle	20
Figure 14:	Dataglove	20
Figure 15:	SOFT Architecture	21
Figure 16:	Connecting to a Virtual World	28
Figure 17:	CP Architecture	33
Figure 18:	Repo-3D Architecture	37
Figure 19:	Use of Server for Reliability	45
Figure 20:	Scalability	45
Figure 21:	Categories of Multicast Applications.	53
Figure 22:	First Working Prototype	56
Figure 23:	Synthesizing a Common Environment	57
Figure 24:	Spiral Model Implementation	58
Figure 25:	Phase One	60
Figure 26:	UML Sequence Diagram (Phase One).	61
Figure 27:	Phase Two	62
Figure 28:	UML Sequence Diagram (Phase Two).	64
Figure 29:	Phase Three	65
Figure 30:	Parsed Stream	66
Figure 31:	UML Sequence Diagram (Phase Three).	68
Figure 32:	Phase Four	69
Figure 33:	UML Sequence Diagram (Phase Four)	72
Figure 34:	UML Sequence Diagram (Phase Five)	75
Figure 35:	UML Documentation for NSG Node	81
Figure 36:	Trend Analysis with/without DaBP	87

LIST OF TABLES

Table 1:	Supported Flow-Types of Tele-Immersive Applications15
Table 2:	NSG Contention Flags24
Table 3:	Implemented NSG Tags78
Table 4:	Designed NSG Tags78
Table 5:	NSG Fields.79
Table 6:	Average Server Overhead with DaBP85
Table 7:	Average Server Overhead without86

LIST OF ACRONYMS

ADU	Abstract Data Unit
AI	Artificial Intelligence
AO	Application Object
API	Application Programming Interface
CORBA	Common Object Request Broker
COTERIE	Columbia Object-Oriented Test-Bed for Exploratory Research in Interactive Environments
CP	Community Place
DaBP	Dial-a-Behavior Protocol
DCOM	Distributed Component Object Model
DR	Designated Receiver
DS	Designated Server
DSM	Distributed Shared Memory
HCI	Human Computer Interface
HTML	Hyper Text Markup Language
JDBC	Java Data-Base Connectivity
JVM	Java Virtual Machine
ML	Mapping Layer
ms	milliseconds
NSG	Network Scene Graph
NTII	National Tele-Immersion Initiative
OMG	Object Management Group
ORB	Object Request Broker
PARIS	Personal Augmented Reality Immersive System
RMTP	Reliable Multicast Transport Protocol
SG	Scene Graph
SGML	Standard Generalized Markup Language
SOFT	Software Framework for Tele-immersion
SSG	Standard Scene Graph
SSS	Simple Shared Script
TAWS	Totally Active Workspace
UID	Unique Identifier
VRML	Virtual Reality Modeling Language
VSCP	Virtual Society Client Protocol
XML	Extensible Markup Language

ACKNOWLEDGEMENT

The writers wish to recognize the professionalism and guidance, and all the support of our advisors. We would like to express our appreciation to Mr. Howard Abrams who helped us debug the C++ client implementation and to Mr. Ryan Brunton for his help in debugging the DaBP implementation. Finally, we thank our families for their devotion and support.

I. INTRODUCTION

A. OVERALL GOALS

Alvin Toffler, in his book *The Third Wave*, forecasts that future white-collar workers will be able to work from home. "Home will become the center of society," but people will still "need face-to-face contact with each other to develop the trust and confidence necessary to work together." [TOFF80]

With the capability of computer systems rapidly increasing, and the swift expansion of computer networks, these predictions are approaching fulfillment. Today this vision is materializing through the Software Framework for Tele-immersion (SOFT) project, National Tele-Immersion Initiative (NTII), and Internet2. Their goals respectively are:

1. SOFT

The goal of the SOFT project is the painless introduction of networked collaboration into single-user computer graphics applications. The applications as well as the platforms they run on can be dissimilar.

2. NTII

The NTII goal is to provide a rigorous test for Internet2 and increase the degree of cooperation between the

research laboratories involved. This will aid advances in virtual reality research [WEB1].

3. Internet2

The goals of Internet2 are [WEB2]:

- Enable a new generation of applications;
- Create a leading edge research and education network capability;
- Transfer new capabilities to the global production Internet.

B. SCOPE OF THIS THESIS

The scope of this thesis is the design and implementation of a network architecture for distribution of generic scene graphs. This network architecture provides a common graphical distributed environment for remote clients, which use various platforms. We examine the most effective structure to support this streamed communication between server and clients. Moreover, we investigate how Java-C++ server/clients can be integrated into the SOFT architecture. We examine multi-tier architecture and multicasting technology in order to transfer the vast amounts of data required for networked graphical applications.

1. Server Design

SOFT will support distributed graphical applications collaborating freely in computer graphics client

communities. Thus, the server must have the following characteristics:

a) Multi-Client

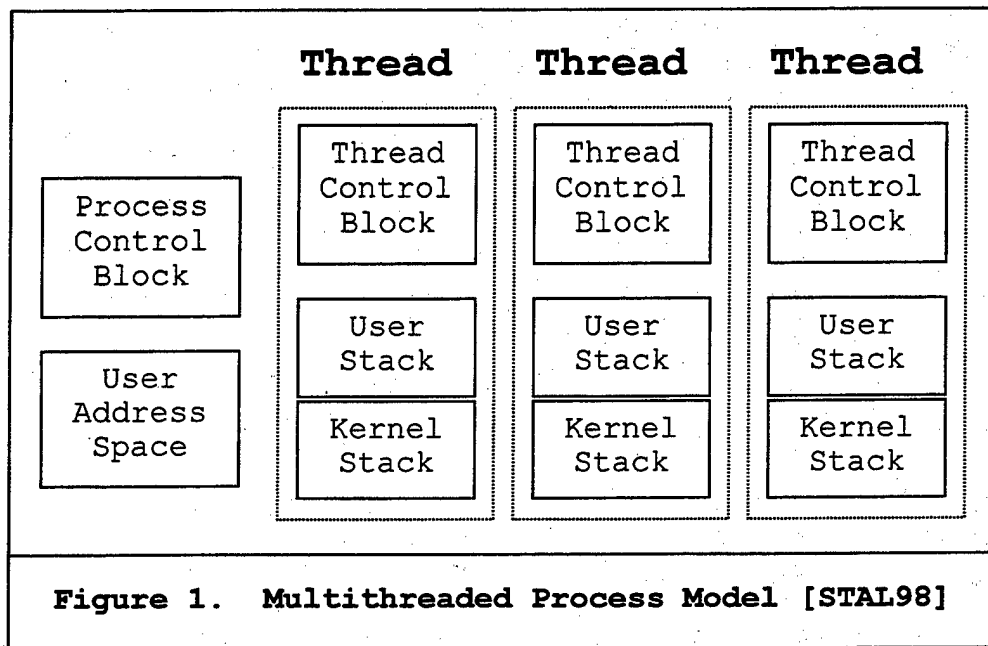
The client/server model can support relationships, such as one-to-one and one-to-many. The one-to-many relationship refers to one client sending information through a server to multiple clients (multi-client).

b) Multithread

A thread is "a dispatchable unit of work." [STAL98] Each thread is executed sequentially and can be interrupted so that a single processor can switch to another thread. Each process can be divided into threads able to run simultaneously while executing an application. Moreover, all the threads can share the state and resources of the same process. This makes multithreading useful for applications that can execute independent tasks which do not need to be serialized. As a result, running multiple threads under the same process can result in less processor overhead because:

- Thread creation requires less time than the creation of a new process;
- A thread can terminate more rapidly than a process;
- Switching between threads is less expensive.

Figure 1 illustrates a multithreaded process model in which each thread has its own control block, user stack, and kernel stack, while sharing the same address space of the common process. An example of this model is a server that listens for and processes multiple clients' requests.



c) Persistence

Two different views of persistence are needed in the SOFT project: scene persistence and persistence binding.

(1) Scene Persistence. The users in the client community will be able to make changes in their graphical environment. However, when they log out and log in again, they must see the scene as they left it. Thus, there

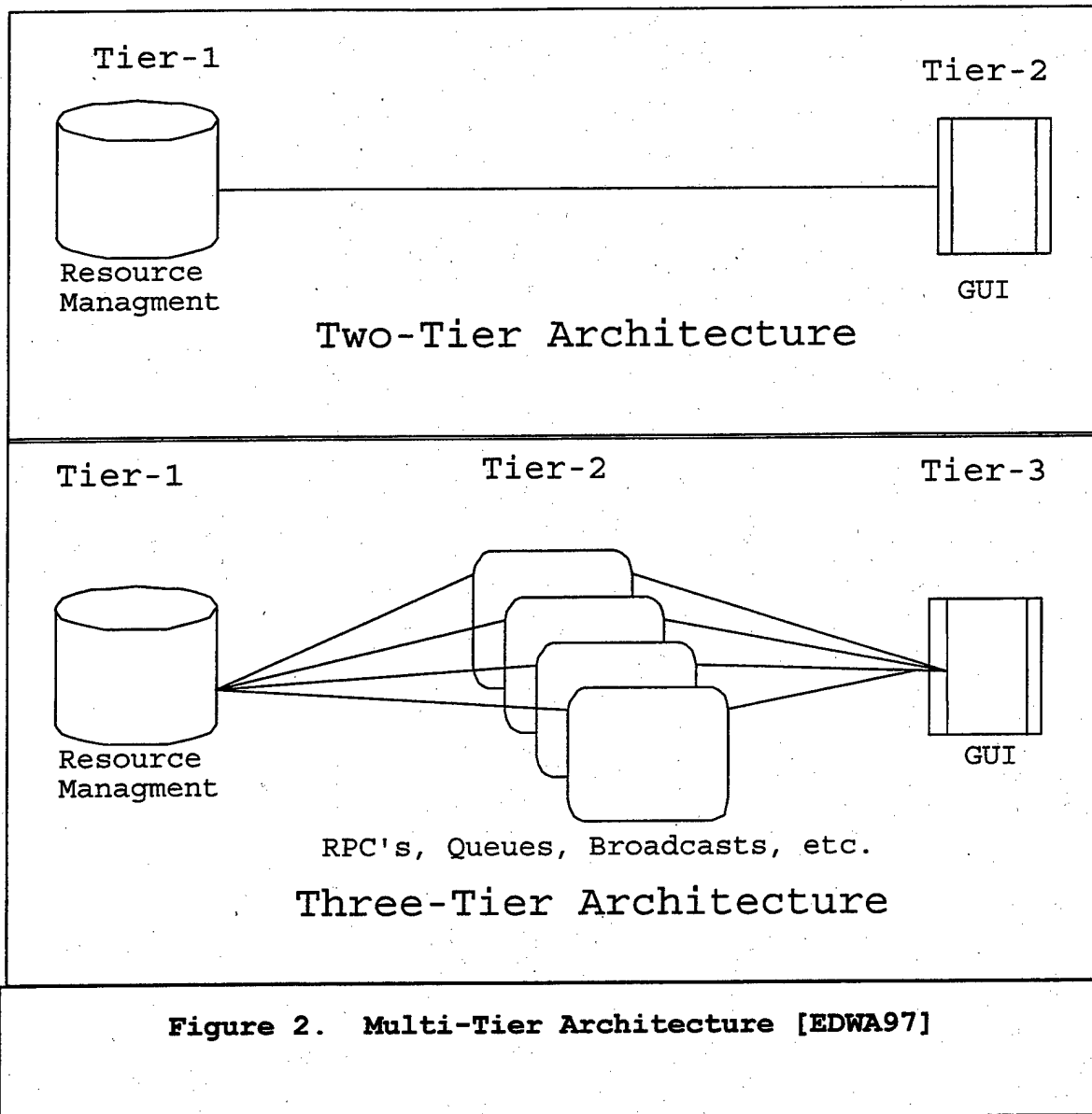
is a need for scene persistence, which must be supported either in the server or the client.

(2) Persistence Binding. When two applications make a logical connection and are prepared to exchange commands and data, they perform a client/server mechanism called binding. The binding mechanism can be either persistent or nonpersistent. The SOFT server must be able to support communication among multiple clients. These repeated calls must be maintained through persistent binding from the server side, which will keep the logical connection open as long as active clients are still in the system.

2. Multi-Tier

The term tier is used to "describe the logical partitioning of an application across clients and servers."

[EDWA97] A two-tier client/server system exchanges information directly between a client and a server. In a three-tier architecture, an application layer between client and server exists (Figure 2). The three-tier architecture is less complex on the client side and has high security, high data encapsulation, excellent Internet support, heterogeneous resource management support, rich communication choices, flexible hardware architecture, and excellent performance. The middle-tier in the three-tier architecture is responsible for servicing requests and responses between the client and the server.



The middle-tier components are of two types:

a) Services

These are stateless procedures.

b) Objects

These have methods and can communicate across networks, operating systems, and languages. They are called Object Request Brokers (ORB's) and support:

(1) Stateless Objects. These do not have a unique state. Microsoft's Distributed Component Object Model (DCOM) is one example and Common Object Request Broker Architecture (CORBA) of Object Management Group (OMG) is another.

(2) Stateful Objects. These use a unique object identifier in order to request services of a certain object.

C. CONTRIBUTIONS AND GOAL OF THIS THESIS

In a shared graphical environment with multiple clients, maintaining the same view for all clients networked together is generally the vital issue. Today, popular graphics libraries, such as OpenInventor, VRML, Java3D, and Fahrenheit use a scene graph as their main data structure. A client in a distributed graphical environment may wish to use any of these various scene graph implementations. SOFT envisions sharing via the scene graph. Each client is able to modify his or her own scene graph. Through network collaboration, this scene graph as bus metaphor provides the changes to other clients for a shared common view. The

primary implementation contribution of this thesis is the server, which efficiently allows for reliable messaging, occasional reduction in traffic, and persistent storage.

D. CHAPTER SUMMARY

The remainder of this thesis consists of the following chapters:

- Chapter II: SOFT. We provide the objectives of the SOFT project, considering network collaboration and interoperability. We propose clients of SOFT, such as telecubicle, telesurgery, and generic sharing of standalone applications. Moreover, we present the logical architecture for the mapping layers of SOFT.
- Chapter III: Related work. Here we evaluate other research related to and similar to SOFT projects.
- Chapter IV: SOFT network architecture. Here we justify the use of the Java server and its centralized role. We also design the client side and provide a multicasting technique for the network architecture.
- Chapter V: Server design and implementation. The methodology and the phases that followed in designing and implementing the server are discussed.
- Chapter VI: Client side architecture and design. We provide the NSG protocol and its serialization.
- Chapter VII: Empirical study of efficiency. We examine the overhead for the server and communication in example sessions.
- Chapter VIII: We reach conclusions and make proposals for future work.

II. SOFT PROJECT

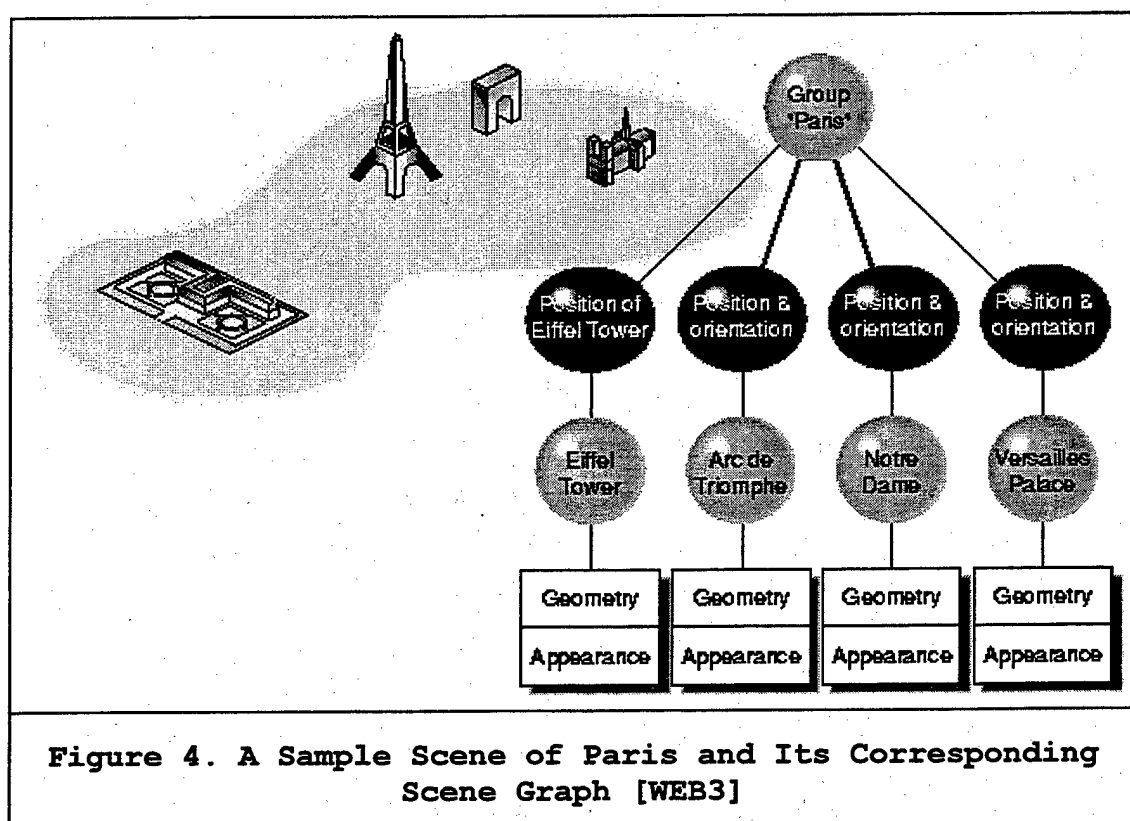
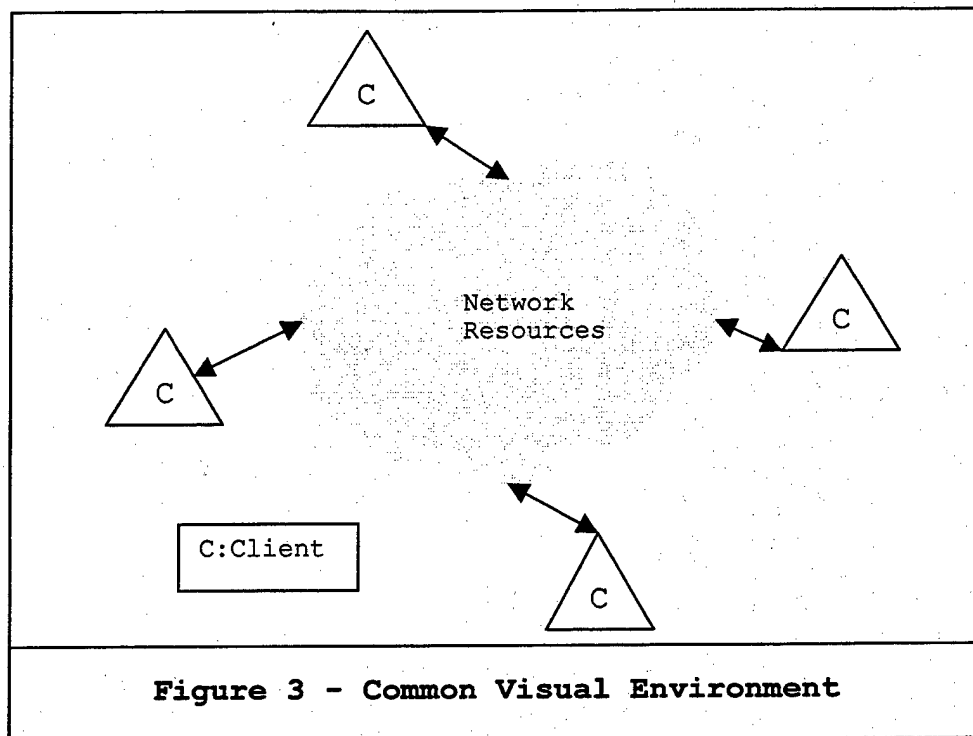
A. OBJECTIVES

As stated by Brown University, the goal of the SOFT project is the painless introduction of networked collaboration into single-user computer graphics applications. The primary research goal is to determine those minimal alterations needed to integrate a stand-alone program neatly into a networked environment.

1. Network Collaboration

In a networked shared environment (Figure 3) each client is able to create his or her own objects and modify any other object, provided it has permission of the owner. Each new client, upon logging in, must be able to share the same view. Additionally, when the user selects an object on his image plane and brings it into his natural working volume, we must decide how this appears to an observer standing close by or at a distance. [PIER98]

A graphical scene may contain various objects, different colors, objects within objects, different levels of detail, etc (Figure 4). These can be defined as a set of nodes organized into a set of hierarchies. The scene graph (SG) is the data structure "used to denote the entire ordered collection of these scene hierarchies." [LEA96]



SOFT will provide the network framework for these SGs (e.g., OpenInventor, Fahrenheit, Java3D, etc.) to communicate. A graphical application using OpenInventor will be able to make calls to another which uses Java3D as SG. Essentially this should lead to free collaboration in computer graphics client communities. The need for access to the application is vital in order to ensure consistency and atomicity in scene modifications. We trade programming complexity for additional collaboration functionality.

a) Layering

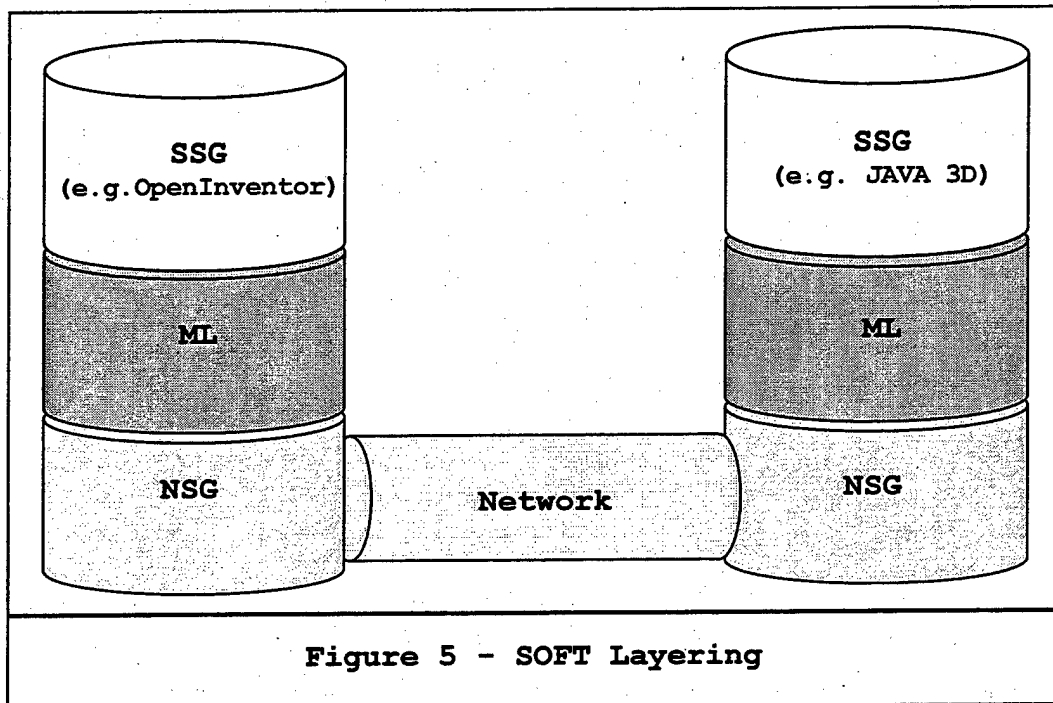
Dividing SOFT into three logical layers reduces this programming complexity. The three layers are:

(1) The Standard Scene Graph Layer (SSG). The SSG is the SG used by a specific application. Any extensions to the SSG will be contained as a sublayer in order to provide the functionality needed for the next layer, the mapping layer.

(2) The Mapping Layer (ML). The ML provides the mapping between the SSG and the Network Scene Graph (NSG). As we may have different kinds of SSGs, architectures, and languages, a new ML has to be written to support each desired combination.

(3) The Network Scene Graph Layer (NSG). The NSG is a special abstract SG that is not displayed, but

rather changes to it are mapped to changes in the SSG. The NSG is shared among all networked hosts in a SOFT session. It must contain data formats and contention resolution mechanisms. Additionally, it must specify the SG data semantics to be implemented in the mapping of the mapping layer. Thus, an API is used indirectly through the mapping layer. The SOFT layering is shown in Figure 5.



b) Sharing via Scene Graph

The NSG is responsible for maintaining the same view for all the clients networked together in the same session. This is achieved as follows:

- Each SG node maps its own set of 3D objects and attributes to the NSG;

- The network layer uses the NSG as a bus metaphor to transmit the created or modified nodes to the remote client's NSG;
- The NSG of the remote client is then mapped to its own SG.

B. PROPOSED CLIENTS OF SOFT

SOFT aims to support any kind of distributed graphical application. Such applications can be, for example:

- Telecubicle (office of the future, enhanced teleconferencing);
- Medical Diagnosis and Telesurgery;
- Surveillance;
- Bomb Disposal;
- Mining.

We focus on the telecubicle because some industrial progress has been made in this area [WEB4]. Also, some significant progress has been made in telesurgery, although it is too early for a commercialized product.

1. Telecubicle

As synthesized by Andy van Dam, a pioneer of graphics at Brown University, immersion and presence can be independent. People surrounded by panoramic views will feel immersion without presence. [DERT98] Telecubicle will be a new interface design of an office that can appear to become

one quadrant in a larger shared virtual office space. As shown in Figure 6, a telecubicle will consist of a stereo-immersive desk surface and at least two stereo-immersive walls. Projectors on the ceiling and various other points will render the images on the walls with ultra high color resolution (i.e., 5000x5000). Moreover, a telecubicle [DeFA98]:

- Will be stereo capable without special glasses;
- Will be networked to teraflop computing via multi-gigabit networking with low latency;
- Will be available in a range of compatible hardware and software applications;
- And will incorporate AI-based predictive models to compensate for latency and anticipate user transitions.

Tele-immersive applications need to support nine distinct flow types [FOST98], which are shown in Table 1. Telecubicles will be linked to form a common work environment, demanding the previous flow of information. As shown in Figure 7, workers will be able to share virtual objects and data while having eye contact with each other. Henry Fuchs believes that "the ceiling lights are replaced by computer controlled cameras and 'smart' projectors that are used to capture dynamic image-based models with imperceptible structured light techniques, and to display

high-resolution images on designated display surfaces." [FUCH98]

Table 1. Supported Flow-Types of Tele-Immersive Applications

Flow type	Latency	Reliable	Multicast	Stream
Control	< 30 ms	Yes	No	No
Text	< 100 ms	Yes	No	No
Audio	< 30 ms	No	Yes	Yes
Video	< 100 ms	No	Yes	Yes
Tracking	< 10 ms	No	Yes	Yes
Database	< 100 ms	Yes	Maybe	Maybe
Simulation	< 30 ms	Mixed	Maybe	Maybe
Haptics	< 10 ms	Mixed	Maybe	Yes
Rendering	< 30 ms	No	Maybe	Maybe

A goal of telecubicle technology is to give the scientist, the engineer, and the worker the impression of collaboration. People will interact with each other through this mix of real and virtual environment - manipulating objects, experimenting, simulating, designing, amusing, and teaching.

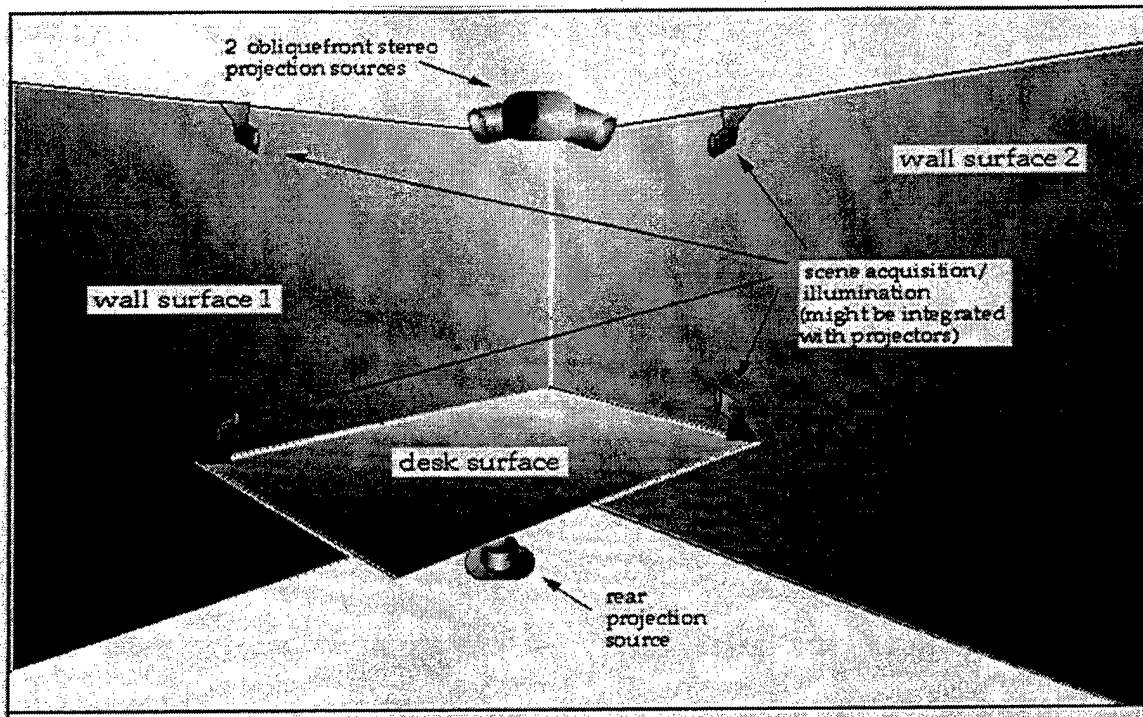


Figure 6. Components of a Telecubicle [WEB5]

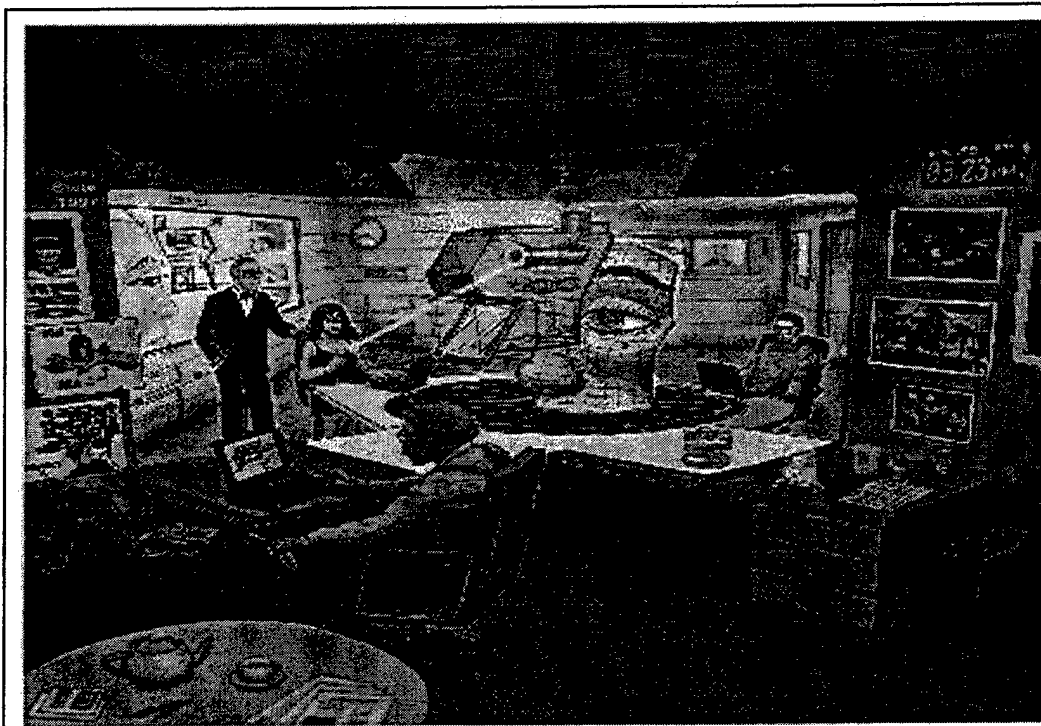
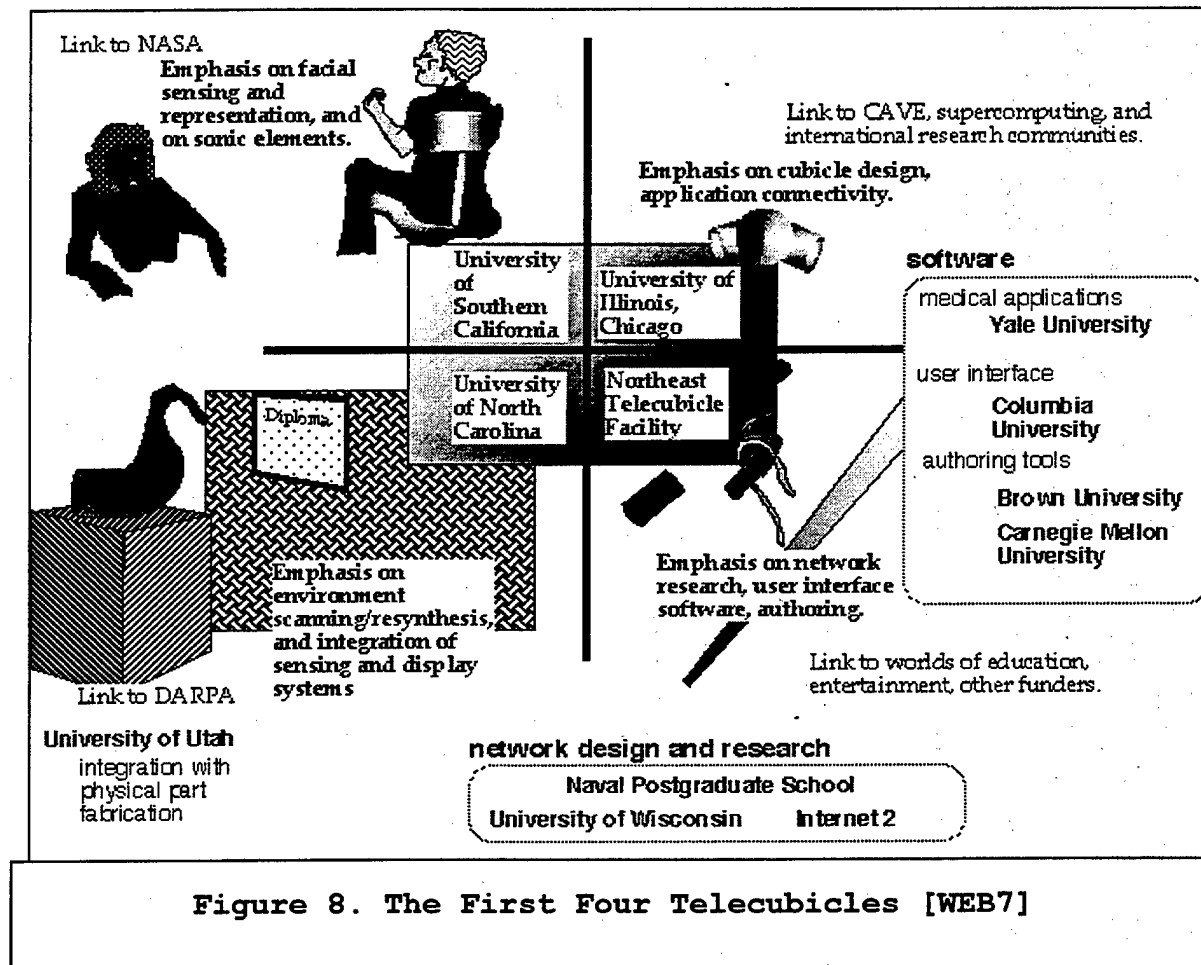


Figure 7. Intuitive Human Communication [Andrei State]

Figure 8 shows the first four telecubicles that have been connected. [WEB7]



Tele-immersive work stations, such as CAVE (Figure 9, [WEB8]), ImmersaDesk3 (Figure 10, [WEB9]), Totally Active Workspace (TAWs), and Personal Augmented Reality Immersive System (PARIS), have already been implemented [DeFA98]. During her nomination as Indiana University's first Distinguished Visiting Technologist in its Advanced Information Laboratory at University, Professor Donna J. Cox

described this process as "stepping inside the computer, because it widens the user's creativity and control [WEB10]."

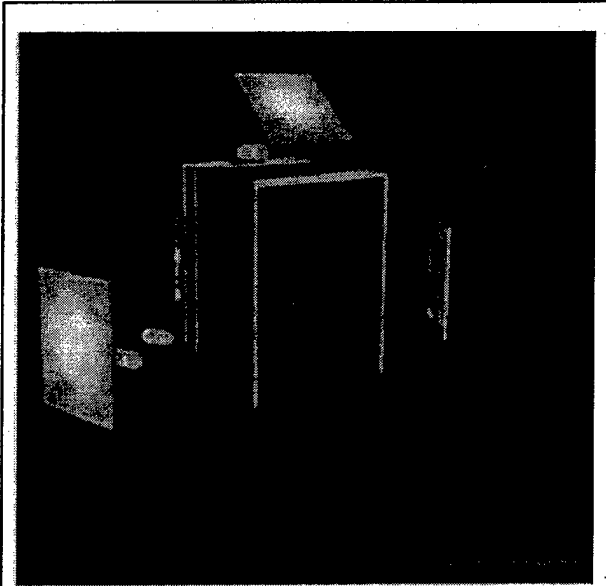


Figure 9. CAVE [WEB8]

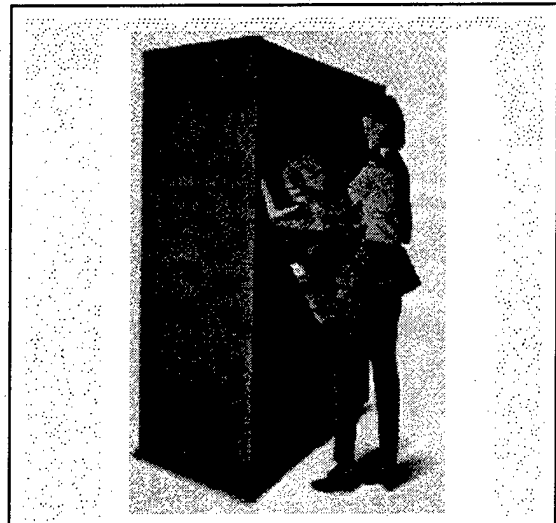


Figure 10. ImmersaDesk [WEB9]

2. Telesurgery

A. Alexander in 1978 [ALEX78] was one of the first researchers to formulate the concept of telesurgery. "The driving force behind military interest in remote surgery is that 90% of all deaths in wartime occur on the battlefield, before surgical care can reach the casualty" [HILL97]. Telesurgery can save the lives of isolated patients in the battlefield, rural areas, and aboard ship.

With telesurgery, the surgeon can perform surgery remotely (Figure 11, [WEB11]) using a special telesurgery console (Figure 12, [WEB12]), instruments (Figure 13, [WEB13]), and haptic devices (Figure 14, [WEB12]).

But surgery usually involves many other people (other surgeons, medical staff, and observers) all needing to share information. These people must also share the same view in telesurgery. This can be achieved through the network collaboration between clients. Each can interact and simulate face-to-face cooperation. The major components of telesurgery are [WEB14]:

- Human Computer Interface (HCI);
- Computer Assistance;
- Communication Methods;
- Telesurgical Worksites.

SOFT could benefit the "communication methods" component by providing the networked collaboration among clients, and the HCI by providing an identical view to everyone.

3. Generic Sharing of Stand-Alone Applications

Stand-alone applications, already written in popular scene graph implementations, should be able to use SOFT for networking capabilities with minimum effort. This will

increase reusability and performance of graphics stand-alone applications.

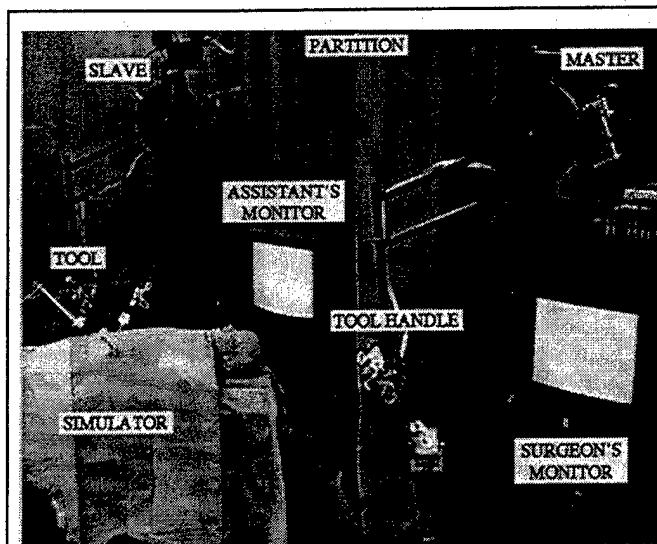


Figure 11. Telesurgery System Components [WEB11]



Figure 12. Telesurgery Workstation [WEB12]

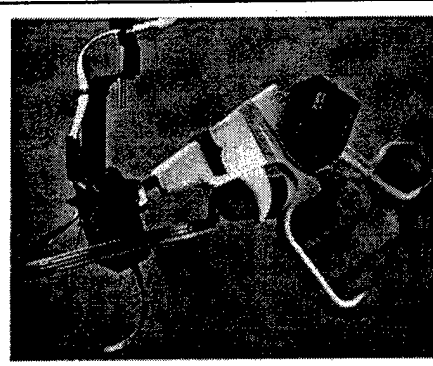


Figure 13. Master Tool Handle [WEB13]

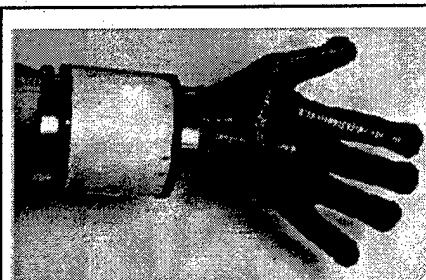
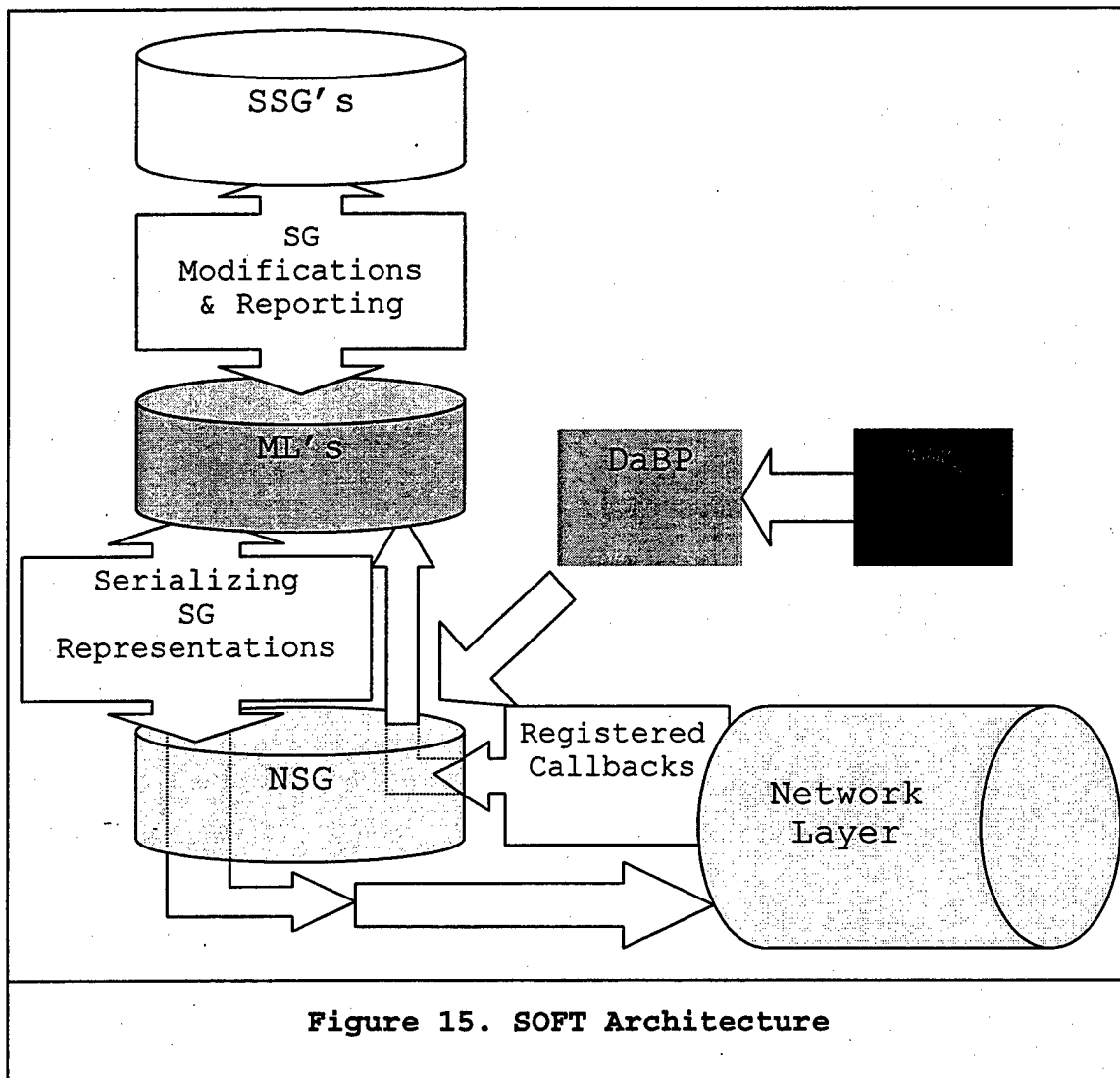


Figure 14. Dataglove [WEB12]

C. ARCHITECTURE

Tele-immersion is likely the most technically challenging advanced network application. Thus, considering interoperability and standardization, the architecture shown in Figure 15 has been chosen to support the previously identified nine distinct components to tele-immersion architecture, such as flow control, audio, video, tracking, etc.



1. SSG and Reporting

SOFT is designed to be independent of SG implementations. OpenInventor has been chosen as the primary SG, and Java3D and Fahrenheit will likely follow. Implementation has begun in C++ on SGI and SUN machines. SSG will be composed of nodes and fields with a unique identifier (ID) for each host. The SSG will report any modifications to the ML. Every modification requires locking for consistency purposes.

2. Mapping Layer (ML)

The ML is a single-base mapping layer between the SSG and the universal NSG. Being the interface between the SSG and the NSG, the ML accepts callbacks from the NSG. The ML behaves as follows:

- Establishes top level reporting and defaults, initializing the NSG and network;
- Recognizes SSG changes and serializes them;
- Passes changes to the NSG;
- Registers callbacks from the NSG;
- Deserializes NSG information and reconstructs the respective SG nodes (special nodes with no analogy in the supported SG are simply ignored);
- Passes deserialized SG nodes to the SSG.

The ML will require an API for locking objects on the network for editing. This API cannot be in the NSG layer because it must translate an SSG node to enable locking of the appropriate NSG node.

3. Network Scene Graph Layer (NSG)

The NSG maintains a common hierarchy of nodes available to the client community. Any ML modifications are mapped to the NSG and reported to the other clients. This is implemented through an API, responsible for exchanging information between the ML and the NSG. The NSG has the following structure:

a) NSG Node

A node is the fundamental element of a scene graph, such as separator, color, cone/sphere, etc. The NSG node supports any type of node. To do so, it contains certain fields.

b) NSG Node Fields

- Node type;
- Unique identifier for the node, namely a node ID and a timestamp;
- Unique identifier for the creator/owner of the node, i.e. the correspondent IP address and the communicating port;
- A set of contention flags.

c) NSG Field Contention Flags

Proper handling of contention issues requires a network token/locking/check-out mechanism. The desired behavior is integrated into the system with the following flags to this point (Table 2).

Table 2. NSG Contention Flags

Flag	True	False
Distribute	If (own object) Distribute locally Else Send to owner	Do not share writes
Local edit	Set value on local write	none
Remote edit	Set value locally when reading network change	none
Local callback	Invoke a list of local callback when reading local change	No callback on local change
Remote callback	Invoke a list of remote callback when reading local change	No callback on local change

This architecture benefits in hiding network proprietary issues. Additionally, it gives the opportunity to various SGs to call each other, even over different hardware platforms.

4. Dial-a-Behavior Protocol (DaBP)

Fundamentally, a protocol is used for exchanging data. This data exchange is a crucial issue in the SOFT architecture. Serializing and deserializing the exchanged data among clients over the network is a very cumbersome approach. With this approach, whenever there is a change in the data format of the exchanged data, the programmers must rewrite and recompile the application. For such a large project as SOFT is, flexibility was needed. This flexibility is provided through the DaBP, which provides methods to return the data from the network packets.

The DaBP, developed at the Naval Postgraduate School, is currently implemented with Java and there is a C++ version under development. In this protocol, we must specify the field names and types of the exchanged data. This information is specified in a structured text file, using the Extensible Markup Language (XML), which is a reformulation of Standard Generalized Markup Language (SGML). XML is more powerful than Hypertext Markup Language (HTML), a standard SGML subset, because it can identify and process document structure and provide extensibility [SURV99]. Moreover, XML allows users to define their own tags and customize the way the protocol behaves.

DaBP is described with a predefined set of fixed tags. Each tag is uniquely associated with a specific field in the XML file. Beyond this, the protocol interpreter is responsible for converting binary data into usable information. Consequently, with the use of this protocol, we achieve the required flexibility. Thus, the NSG layer does not have to serialize and deserialize the exchanged data. Instead, the NSG layer retrieves the data fields needed, querying the packets using DaBP methods.

III. RELATED WORK

A. INTRODUCTION

The spread of software systems, the expansion of networks, and the relatively low cost of graphics cards and their increased performance has generated the need for network collaboration among graphical applications. Additionally, the variety of important standalone graphical applications led developers to consider reusability and networking carefully. Concerning these two factors, the related work is divided into three areas of interest:

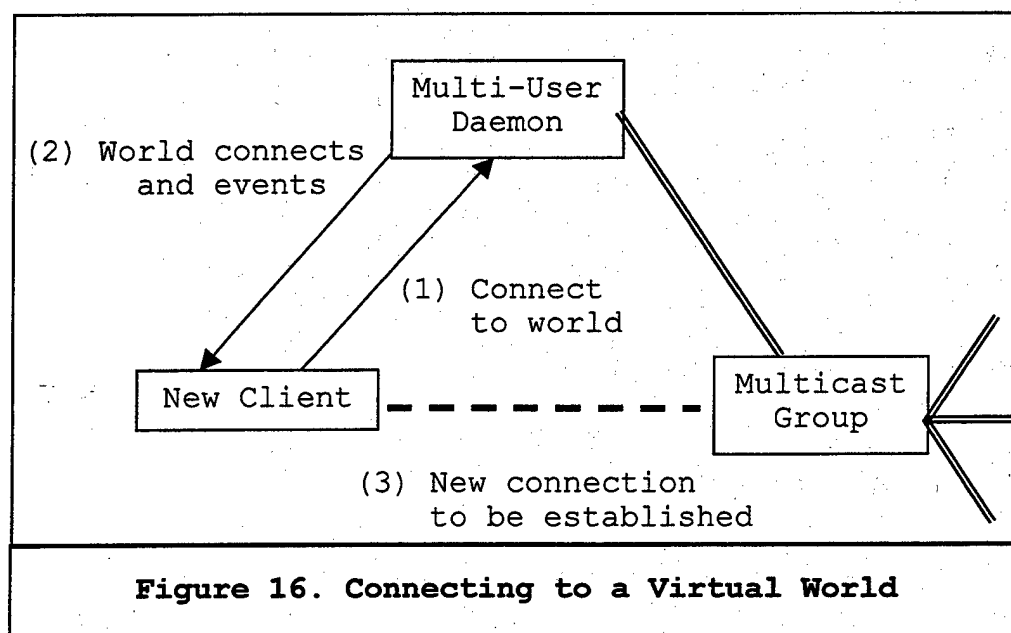
- Interconnecting proprietary stand-alone applications;
- Interconnecting heterogeneous virtual environments;
- Building frameworks for distributed virtual environments based on client/server approach and replication.

B. INTERCONNECTING PROPRIETARY STAND ALONE APPLICATIONS

The popularity of the Virtual Reality Modeling Language (VRML) used to describe 3D worlds established it as a language which developers could use for graphic applications. The great progress in the presentation of stand-alone virtual worlds encouraged the VRML community to implement multiple worlds and users to interact with each

other. This progress was made possible by the network capabilities of the Java language, which provides appropriate and efficient communication mechanisms.

Wolfgang Broll presented an approach in "Supporting Multiple Users and Shared Applications with VRML" [BROL97]. He exhibits a network infrastructure and appropriate mechanisms to partition the virtual worlds. He uses a multi-user daemon, which provides reliable TCP/IP connections in order to download shared virtual worlds. To overcome the problem of inconsistency between shared worlds, the daemon retains incoming events and transmits them to new clients after the transmission of the basic virtual world has been completed (Figure 16.) After the successful transmissions, the TCP/IP connection is closed.



The multi-user daemon acknowledges all events sent by consecutively numbered messages. The sender resends all messages that haven't been acknowledged. A unicast connection is used for receiving the missing messages. To avoid the multi-user daemon bottleneck caused by unicast connections, additional server daemons may be used. However, to avoid slowing down the performance of multi-user daemons, all the additional servers require access to the multicast group.

Usually, in a large-scale virtual environment, the client needs to participate in only a portion of the world. According to Broll, the solution is to subdivide the world in regions and link them through mechanisms that do not interfere with user navigation. Thus, only currently visible parts of the world need to be updated, reducing the network traffic. Broll's approach exploits the VRML Region node. Moreover, he uses the VRML User node and the VRML Avatar node to limit user interactions in the shared virtual environment. Each user has a personal VRML file, which is transmitted by the local browser to all the participants and the central daemon. In this way, each participant has his or her own individual scene graph. Furthermore, shared behaviors and interactions between clients are required.

Broll identifies four general classes of shared behaviors:

- Autonomous behaviors (e.g. a clock, a blinking light);
- Synchronized behaviors (e.g. a bouncing ball, a flying bird);
- Independent interactions (e.g. a button to ring a bell);
- Shared interactions (e.g. avatars with a single region where the behavior is described within this single region).

He represents these shared behaviors and interactions with nodes. Behavior nodes receive and send events. Finally, he uses synchronization and locking mechanisms for the events.

C. INTERCONNECTING HETEROGENEOUS VIRTUAL ENVIRONMENTS

Capps et al. in their work "Distributed Interoperable Virtual Environments" present the use of a software bus (Polylith) as a utility to compose existing applications instead of modifying them [CAPP96]. They focused on the issue of interoperability between different graphic software applications that run on different hardware platforms. They assumed that next generation virtual environments will not

depend on specific languages and hardware. Furthermore, they were concerned about reusing software components.

Based on these issues, they used a Polyolith software bus as a connection module, capable of listening to a variety of event classes and handling them both simultaneously and asynchronously. They built an event-listener as a central intermediary that can serve all the components in the virtual reality environment. This intermediary provides the "glue" for interconnecting the two different virtual reality applications.

This software bus encapsulates decisions concerning the interfacing of modules instead of distributing them among the participants. Thus, there is only one responsibility for mapping each domain into the abstract bus specification. Then developers for either side can easily use these abstractions within their configurations.

These researchers conclude that the same collaboration issues used in shared editors and workspace environments apply in multi-user virtual environments. These can be exploited by software bus modules that provide for the mapping between the interconnecting domains.

D. BUILDING FRAMEWORKS FOR DISTRIBUTED VIRTUAL ENVIRONMENTS BASED ON A CLIENT/SERVER APPROACH AND REPLICATION

We present the following three different approaches using a client/server mechanisms and replication:

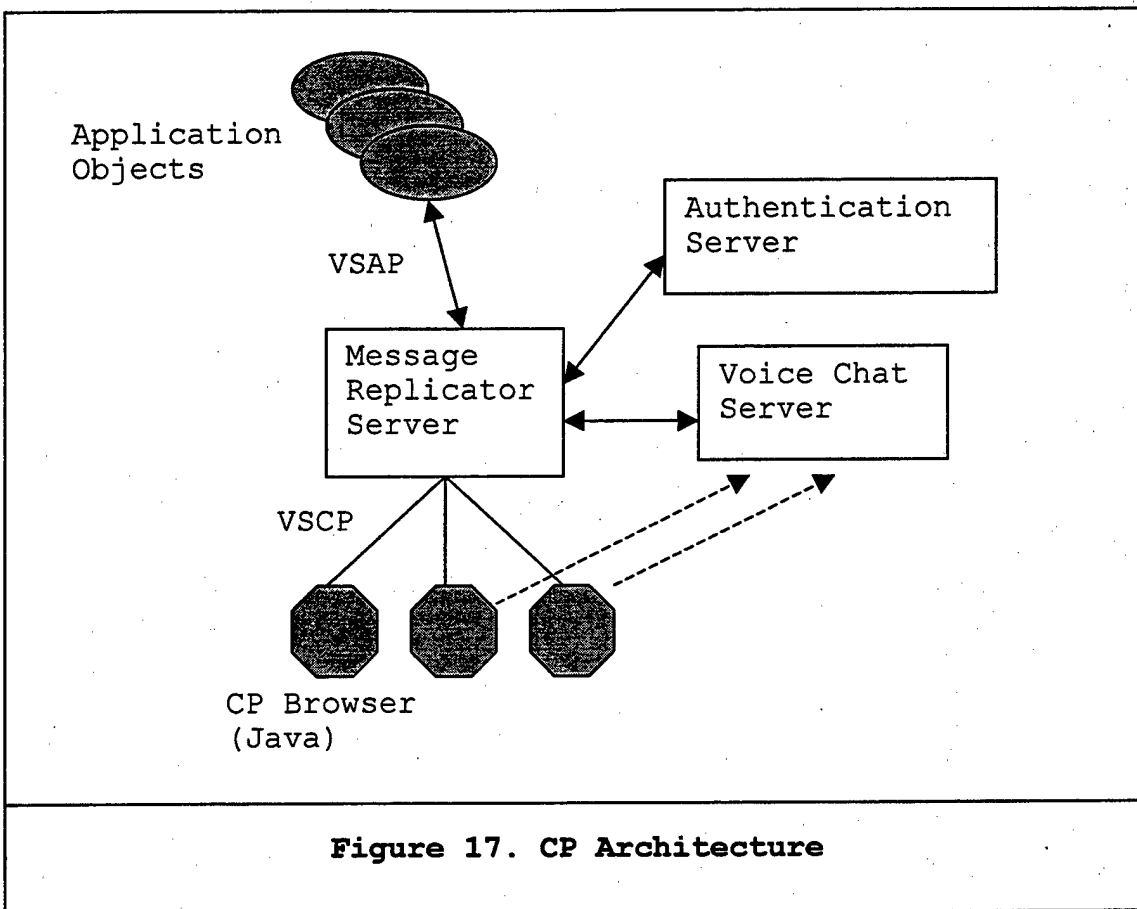
1. Community Place (CP) Architecture

Lea, Honda, and Matsuda in their work "Community Place: Architecture and Performance" [LEA97] presented a client/server system where a central server is responsible for sharing a VRML scene. Participants connect to this server in order to load the VRML file that is responsible for rendering the shared scene. The basic architecture is shown in Figure 17. The order of actions is:

- The Community Place (CP) browser loads the 3D data file (VRML format);
- The CP browser contacts the server via the Virtual Society Client Protocol (VSCP) that runs above IP;
- The server informs the CP browser for other participants.

The VSCP runs above TCP and ensures connectivity. It has an object-oriented packet definition allowing applications to extend the basic packet format with application specific messages. Thus, VSCP ensures exchanging of script level messages that permit browsers to share

events and therefore support shared interaction with the 3D scene.



Each user navigating through the scene sends position information to the server. Using area of interest algorithms, the server decides which other browsers a user needs to be aware of. The server need not know information about the scene loaded by the browser.

Lea, Honda, and Matsuda introduced the idea of a Simple Shared Script (SSS) model mechanism, which is responsible for downloading the same script and executing it locally. To

deal with issues such as ownership and persistence, they added the notion of a master browser to the SSS model. Whenever the master browser is selected by the server, it is told that the master has been selected. Scripts are capable of sending events to other browsers. In cases where the user wants to implement serialization as a scene object, it is done via the master browser. Afterwards, the master browser distributes the changes.

For more complex situations, they introduced the idea of an Application Object (AO) which exists externally to the browser and the server. This AO consists of three parts:

- The 3D data description that represents the application in the shared scene;
- The associated scripts that accept user input and communicate back to the AO;
- The AO side code that implements the application logic.

The AO allows the creation of 3D objects dynamically during run-time. The applications use the Virtual Society Application Protocol (VSAP) to register their application objects with the server.

They use a spatial model to reduce network traffic among participants who share a 3D scene. Their communication mechanism is based on multicasting.

2. Spline Architecture

Another approach for building these frameworks is the Spline architecture used for the *Diamond Park Virtual Reality System* designed by the Mitsubishi Electrical Research Laboratory (MERL) [WEB15]. Spline's world model is not a scene graph but rather an object-oriented database supporting visual and audio information. The objects have ownership attributes to avoid reader/writer conflicts. The ownership of an object can be transferred from one process to another. Spline's objects do not persist over time.

This architecture is based on replication so that each world model resides in each application process. Focusing on the issue of speed Spline provides approximate equality between world model copies. Users are grouped together in locales of interest. Each locale is associated with a separate multicast communication channel avoiding propagation of messages to uninterested participants. A hybrid communication approach was proposed for the Spline 3.0 version where client/server communication will be point-to-point and server/server communication will be peer-to-peer multicast. The exchanged messages are divided into small rapidly changing objects, large slowly changing objects, and continuous streams of data. In Spline 3.0, Java will be the primarily high level interface.

3. Repo-3D Architecture

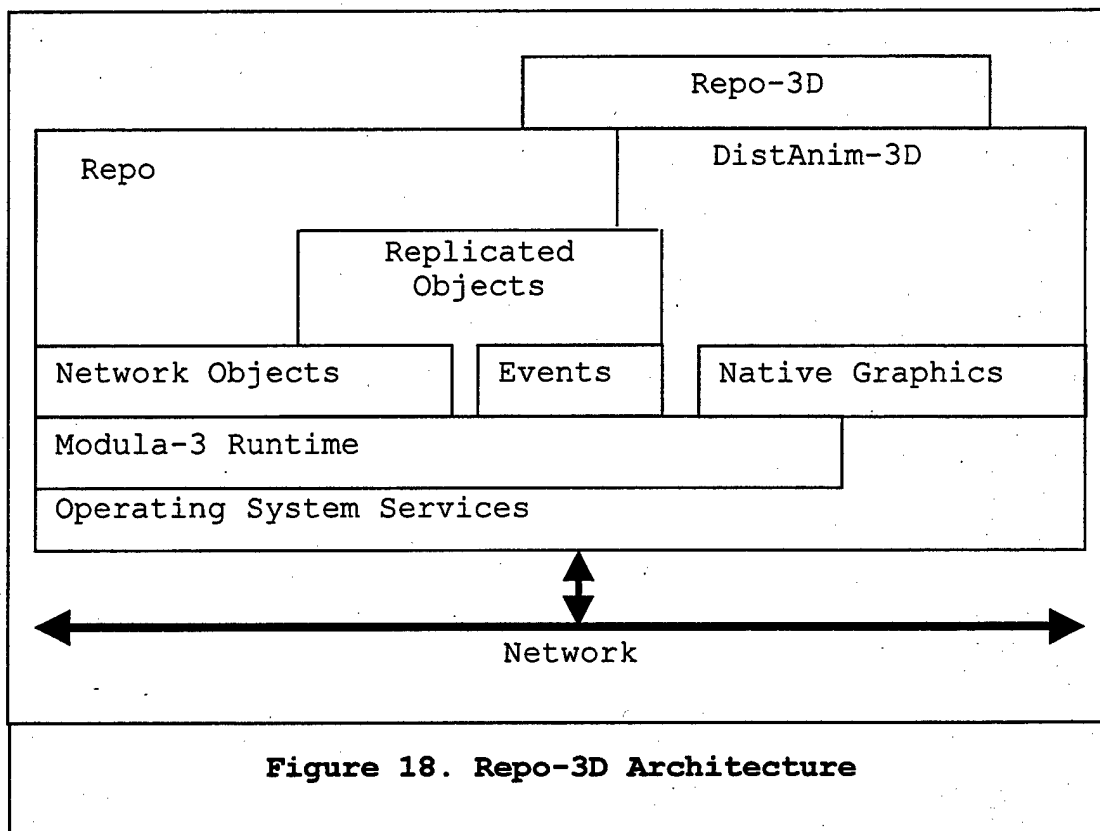
Blair MacIntyre and Steven Feiner in their paper "A Distributed 3D Graphics Library" present Repo-3D, a general purpose object-oriented library for developing distributed, interactive 3D graphics applications across a range of heterogeneous workstations [MACI98]. According to them, from the programmer's viewpoint, objects reside in a large distributed shared memory (DSM) instead of a single process. The underlying system is responsible for replicating the objects among the processes. Simple, remote, and replicated are three types of distributed objects semantics in DSM.

Since the refresh rate is a crucial factor in interactive-graphics virtual-reality applications, the data needs to be local to the process doing the rendering. Repo-3D uses the Columbia Object-Oriented Testbed for Exploratory Research in Interactive Environments (COTERIE) as the replication mechanism because neither Inventor nor Java3D provides support for distribution.

The CORBA solution was rejected for being too heavyweight and for not supporting replication. Java proved to be more suitable for the implementation language because of its cross-platform compatibility and support for threads and garbage collection. The Repo-3D architecture is shown in Figure 18 where distributed data sharing is provided by two

packages, the Network Object client/server object package and the Replicated Object shared object package.

Distanim-3D is derived from Anim-3D, a powerful, non-distributed, general purpose 3D graphics library. Anim-3D is a scene graph model suitable to a distributed environment. In Anime-3D, properties are attached to nodes and any changes do not affect the result. Unlike Inventor, ordering is not necessary. Properties are only inherited down the graph.



The Network object package provides support for remote objects. This package is similar to Java's Remote Method Invocation (RMI). The Replicated Object package of course supports replicated objects. Each process can call any method of an object it shares. This package follows the principles of atomicity and serialization of an object.

Repo-3D rationale provides programmers with the illusion of a large shared memory using Distributed Shared Memory (DSM), making it easy for them to prototype distributed 3D graphics applications. Their future work will most likely use Java because even if Java does not support a replication object system, the JSDT may be a fine starting point.

E. CONCLUSIONS

Our SOFT architecture has both similarities to, and differences from, the presented approaches. Like CP, SOFT uses a centralized server.

Its architecture doesn't depend on a particular scene graph (Inventor is used as an example of a scene graph implementation), as the ML is responsible for mapping any scene graph to the NSG. SOFT differs from Spline and CP because it stores scene graphs using Java native structures, instead of an object oriented database. Moreover, SOFT

doesn't depend on any database replication mechanism, as Spline does. Furthermore, instead of distributing VRML files, like CP or Broll's architecture does, SOFT distributes scene graphs.

Lastly, unlike Repo3D, which is based on COTERIE for the underlying distribution of scene graphs, SOFT invokes Java networking methods and is capable of using RMI.

THIS PAGE WAS INTENTIONALLY LEFT BLANK

IV. SOFT NETWORK ARCHITECTURE

A. INTRODUCTION

As outlined in Chapter III, virtual environment researchers are attempting to solve the problem of networking collaborative virtual environments with various architectures. Some developers use multicast groups, others try to solve the problem using replicated databases, and a few others use hybrid techniques. According to Michael Zyda, replicated world databases are more efficient than centralized or distributed shared database schemes, but they generally lack a way to maintain world consistency [ZYDA97]. Also, large virtual environments could use hybrid models with small replicated data sets and a distributed client/server model. Thus, the client/server module could be integrated in such environments.

Thinking of implementation, we considered that Java provides innovative methods for building virtual worlds. Furthermore, since our primary goal was to maintain a common shared view, and no database or replication capability existed, we decided to design and implement a centralized server using Java. Below, we present our decision and provide more details.

B. SERVER

We chose to implement the SOFT network architecture using a centralized server. This server would be responsible for providing a common share view among clients. Below we present both our justifications for the centralized server and the use of the Java programming language.

1. Justification for Centralized Server

First let us examine the rationale for our decision to implement the centralized server. Our decision was based on the following criteria:

a) Reliability

Several data exchanged in a tele-immersion session must be sent reliably. This is required in order to ensure consistency of the world database or to reflect an accurate update from a simulation or user interface event. [FOST99]

As stated in Chapter II, a major aim of SOFT is to provide a common graphical distributed environment, where each client must be able to share the same view. This means that every action on any client must be broadcast to all the other clients concurrently connected. Moreover, users must be able to collaborate using the available objects. Object use must have the permission of the owner and thus a universal locking mechanism must exist. The use of a

centralized server facilitates all these transactions. Clients notify or query the server in order to create, modify, or use an object. The server is responsible for storage, locking, and broadcasting to the currently connected users (Figure 19).

The only drawback is that this kind of implementation is based upon the reliability of a single server. Any failure of this server will affect the common shared virtual environment.

b) Scalability

Usually a server creates a bottleneck, an undesirable factor in terms of scalability. But using multiple servers, we can overcome the bottleneck and increase the scalability. Furthermore, multiple servers can be used in conjunction with the Reliable Multicast Transport Protocol (RMTP). RMTP uses receivers associated with local regions or domains. In each domain, there is a special receiver, called a designated receiver (DR) [PAUL98]. Additionally, a special client can also be a server during a session, called a designated server (DS) (Figure 20).

A centralized server implementation may reveal potential problems that could be spread in a networked collaborative environment. Fortunately, the centralized

server implementation can isolate and solve these problems, providing a more robust scalable module that can more easily be integrated in the previously mentioned architecture.

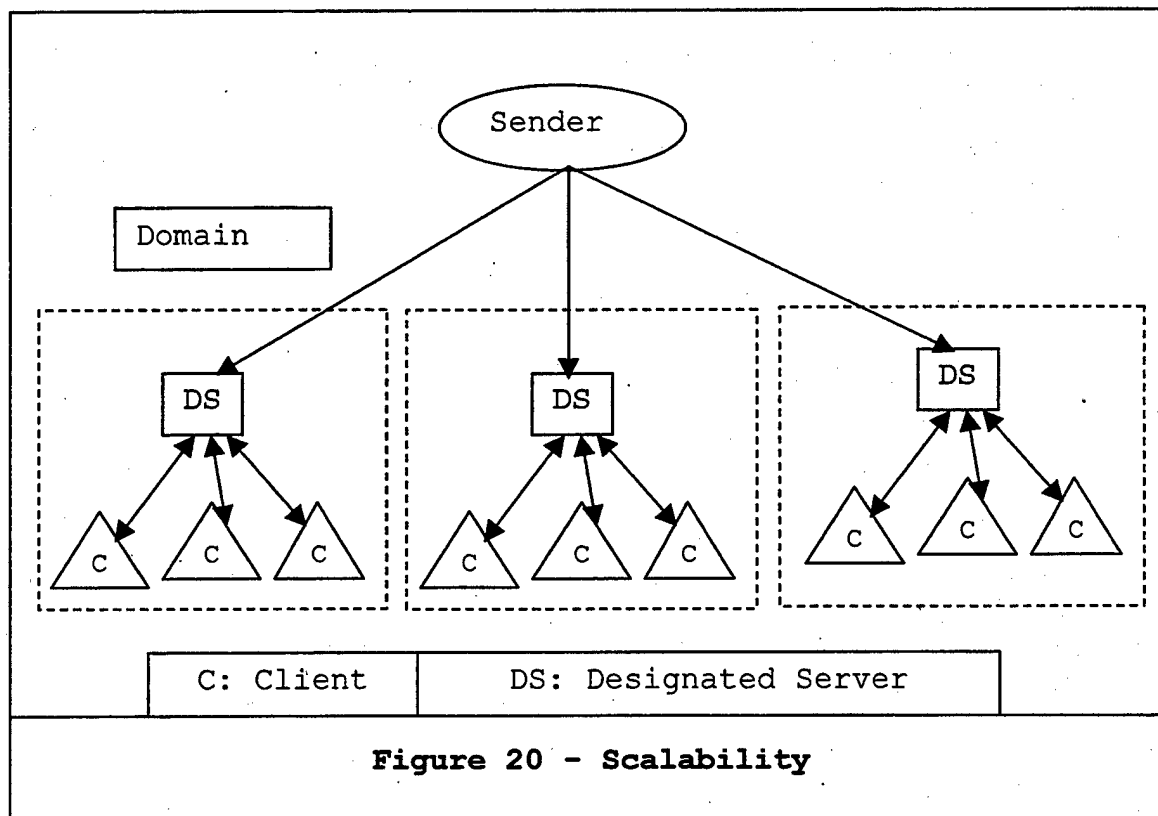
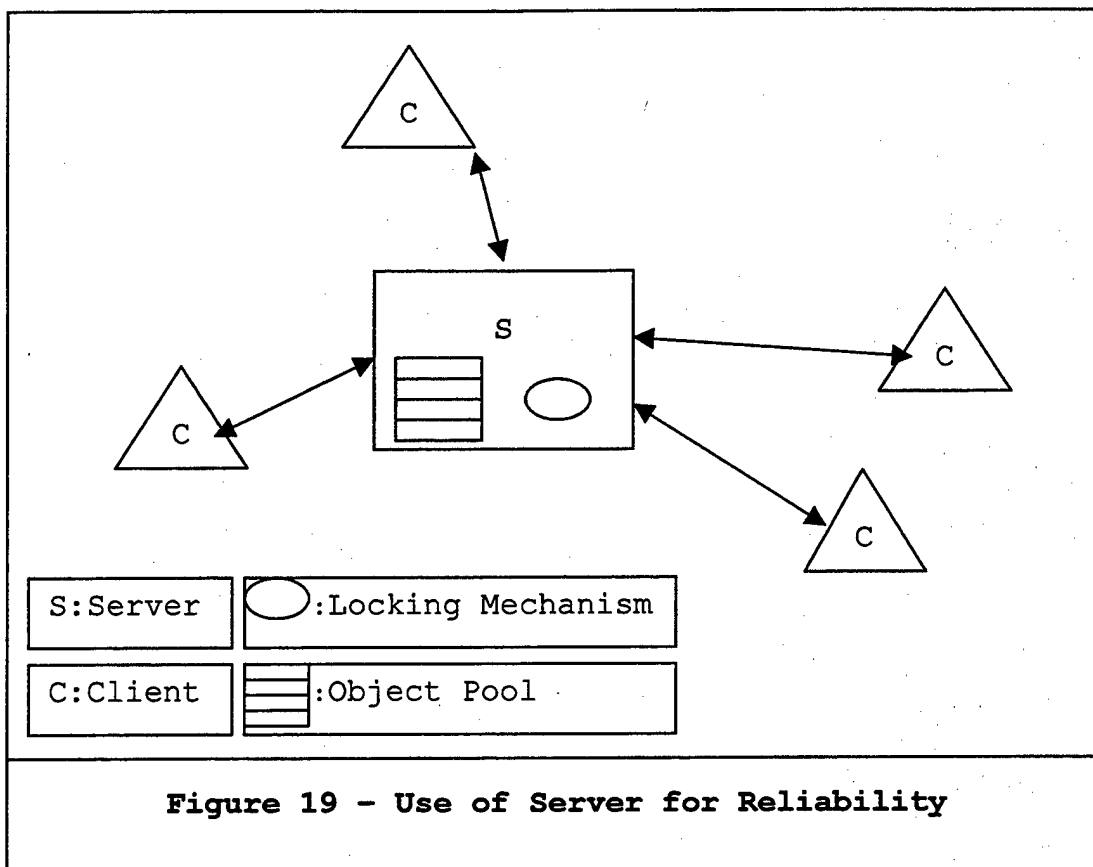
c) Persistence

The status of the objects in a virtual environment varies according to the indication of their contention flags. Some objects will vanish when their owner disconnects from the system, while others may remain until their owner deletes them, even if the owner is not currently logged on the system.

With a centralized server, storing the contention flags and providing the necessary persistence is easy. Also, objects can be independent from client existence and can be retrieved easier from a centralized server.

d) Latecomer Support

When new clients join the virtual environment, they must be aware of all the currently existing objects. The server can store and maintain the current status of every object which is present in the environment. Also, whenever new clients connect to the network, the server sends them all the available data. Thus, the new client becomes a member of the common virtual environment.



e) Conclusion

The centralized server implementation has some drawbacks such as:

- Network bottleneck through which all traffic must pass;
- The environment depends on the reliability of a single machine.

But the benefits are:

- Simple and clear implementation;
- Universal locking mechanism;
- Can be used as a scalable module to serve a domain in a more complex implementation;
- Easier storage/retrieval mechanisms used to provide persistence;
- Latecomer support.

Since the benefits provide a flexibility which is more important in the current state of the project, we chose the centralized server implementation.

2. Justification for Java Server

In Chapter III, the derived conclusions focus mainly on Java implementations. Since the working prototype had been developed in C++ with UNIX as the operating system, we had to choose between C++ and Java. Below we present our criteria

and analysis which resulted in Java being chosen for the implementation.

a) Built-In Standard Libraries

Java contains standard libraries for solving specific tasks [ECKE98]. These tasks include networking and multi-threading, which are major aspects of the server implementation. Using standard libraries promotes rapid development time allows us to focus on the specifications and requirements for the server. On the other hand, C++ relies on third-party non-standard libraries or on code from scratch. Development is time consuming and involves higher risk potential. Moreover, Java standard libraries support database connectivity via JDBC and distributed objects via RMI and CORBA. These features are not of immediate value, but they enhance the scalability and flexibility of the project in the future.

b) Garbage Collection

Memory management during the server session is a factor dramatically affecting robustness. Java provides a built-in mechanism for memory management called garbage collection [CHEW98]. Garbage collection is responsible for non-referenced memory release so that it may be reused. This way memory-leaked-addresses are corrected, and explicitly

deallocating memory is not needed. Note, however, that a memory leak can still exist if unused memory remains referenced. Memory management is a critical factor for the server because there is a permanent "read a stream into a buffer, store the data, release the buffer" loop.

Robust memory management may affect performance too. According to Bruce Eckel, overall, Java could possibly be as fast or faster than C++ [ECKE98]. This can happen because, even though interpreted Java code can be even 20 times slower compared to equivalent compiled C++ code, the new-delete mechanism for memory management in C++ leaves holes in the heap eventually making it slower. The allocation mechanism has to seek available space through those holes in order to prevent running out of heap storage. This searching may seriously decrease performance. The Java garbage collector rearranges memory, allowing the high-speed, infinite-free-heap model to be used while allocating storage. [ECKE98]

c) Platform Independence

Java programs are compiled to an architecture-neutral byte-code format [FLAN97]. These byte-codes can be interpreted by the same version or newer Java Virtual Machine (JVM) on any platform. Platform independence is a

real benefit for the SOFT server as it allows users to use SOFT without purchasing new hardware or installing a new operating system.

d) XML, Java and DaBP

As stated in Chapter II, XML is quite powerful. It provides a universal schema or metadata mechanism for defining, understanding, and interchanging files and data between two systems [SURV99]. XML is already being used by the DaBP. We intend to examine the feasibility of incorporating DaBP in the SOFT architecture. Moreover, "Java is on the XML action both as driver and utilizer of XML capabilities." [SURV99]

e) Evolution - Maintenance

The SOFT project is an evolving environment, and major modifications are likely in the future. Every modification or improvement may affect the server also. A simple Java server will require less effort and time to update than a C++ implementation.

f) Conclusion

The built-in standard libraries for networking, the garbage collection mechanism, the platform independence,

the cooperation with XML and DaBP, and the easiest way to maintain the system prompted us to employ Java.

C. CLIENT SIDE

In the SOFT architecture, the client can be any standalone application (graphical or not). Every client must be able to exchange information among other clients transparently. Therefore, SOFT must provide an API for the upper layers of its architecture.

1. API to Network

The client provides an API for the mapping layer core. This makes the network layer independent of the SSG implementation. The methods available to the mapping layer include the following:

- Initialize the network;
- Set and get per-typed callbacks;
- Serialize data into a single NSG node;
- Get a list of nodes;
- Map from name to NSG node;
- Set and get global callbacks;
- Get and set local root of the scene graph;
- Set, get, and call end of frame callbacks;
- Get list of roots of remote scene graphs.

2. Client High-Level Architecture

The client is an abstraction that provides network access to the mapping layer of the SOFT architecture. It is independent from the SSG and can be used by different graphics applications. It is not hardware specific and handles large and little-endian issues, providing interoperability. The client has been implemented in C++.

D. MULTICASTING

One of the most common ways of communication is one-to-one. The client-server model falls into this category. Another category is unicast communication where the web client retrieves information from the web server. A third example is broadcast communication, for instance radio and television where each client tunes to a certain frequency to retrieve information. Multicast falls between unicast and broadcast and "is a one-to-many communication" [PAUL98].

When providing the same data for multiple graphical applications in the client community, multicast transmission is needed, rather than repeated unicast transmission to each receiver [FOST99]. In this way, the sender can transmit a single copy of a packet, regardless of the number of clients. The routers in the network infrastructure are responsible for delivering the packet to the clients. The

packet is replicated as many times as the number of clients, which improves the efficiency of the network use.

The multicast data can be in a variety of forms: audio, video, haptic device streams, and tracking. The size of the data, especially the video, can be enormous in size per second. Thus, latency plays a key role in the sharing of a common graphical environment among the client communities. So far, the use of multicasting has two drawbacks, which actually do not rely on multicasting itself:

- Difficulty in accessing high-performance multicast enabled networks;
- Lack of middleware tools with access to multicast.

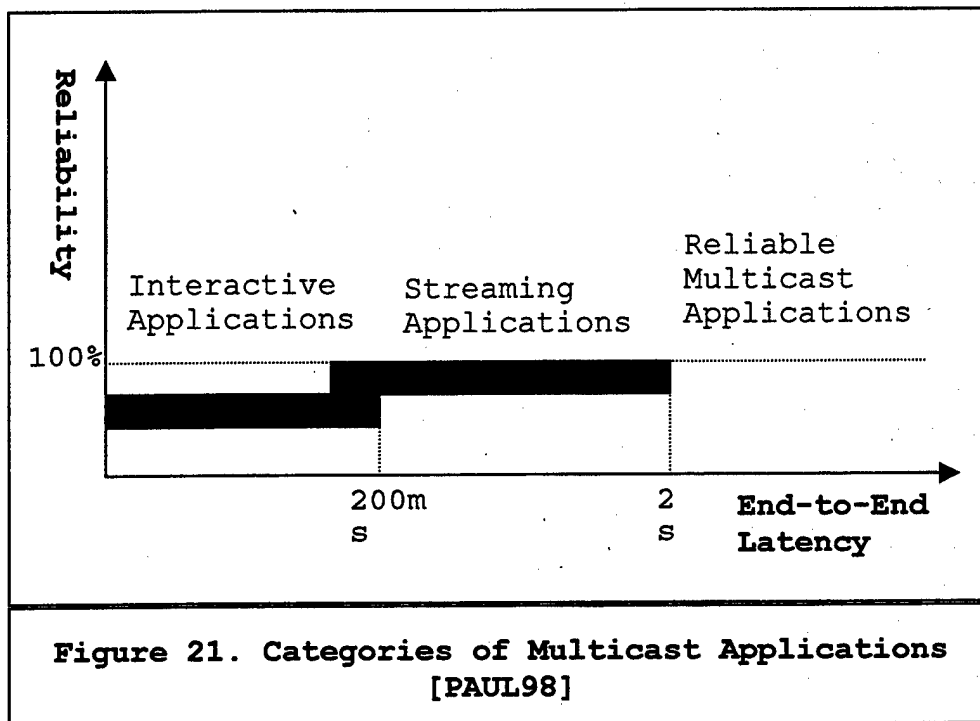
Based on the reliability and the latency requirements, the multicasting applications are divided into three categories [PAUL98]:

- Interactive applications;
- Streaming applications;
- Reliable multicast applications.

The relationship between latency and reliability is illustrated in Figure 21.

For many applications, despite the issues of reliability and latency, there is a basic need for selective

distribution. This means only a certain group of clients must receive the information.



THIS PAGE WAS INTENTIONALLY LEFT BLANK

V. SERVER DESIGN AND IMPLEMENTATION

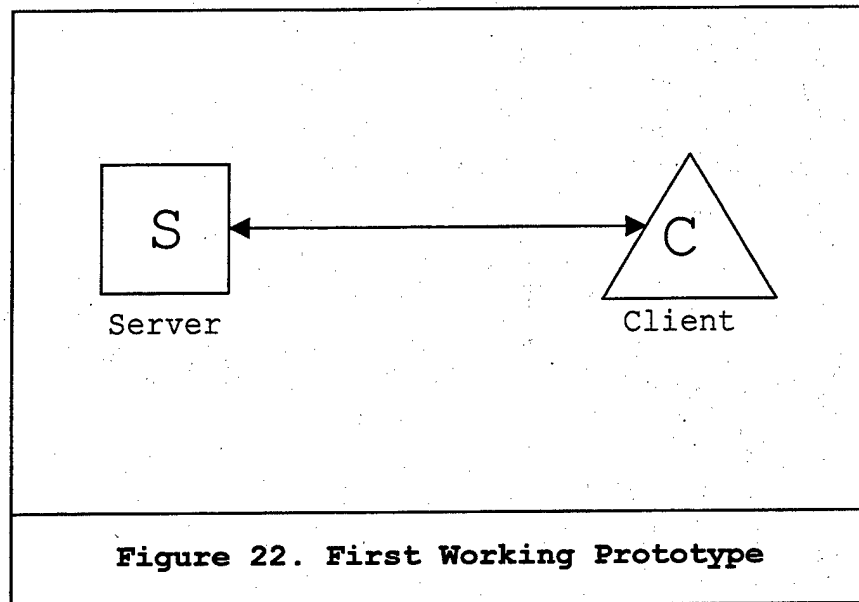
A. INTRODUCTION

In the SOFT architecture, the server must support multiple clients, latecomers, and use the DaBP. The client side could be any standalone application. The server must be responsible for providing the clients a common shared environment. In order to analyze and understand the previous work done, we had to follow a methodology under a strict time frame. We present the methodology used to accomplish this and the way the server was implemented.

B. METHODOLOGY

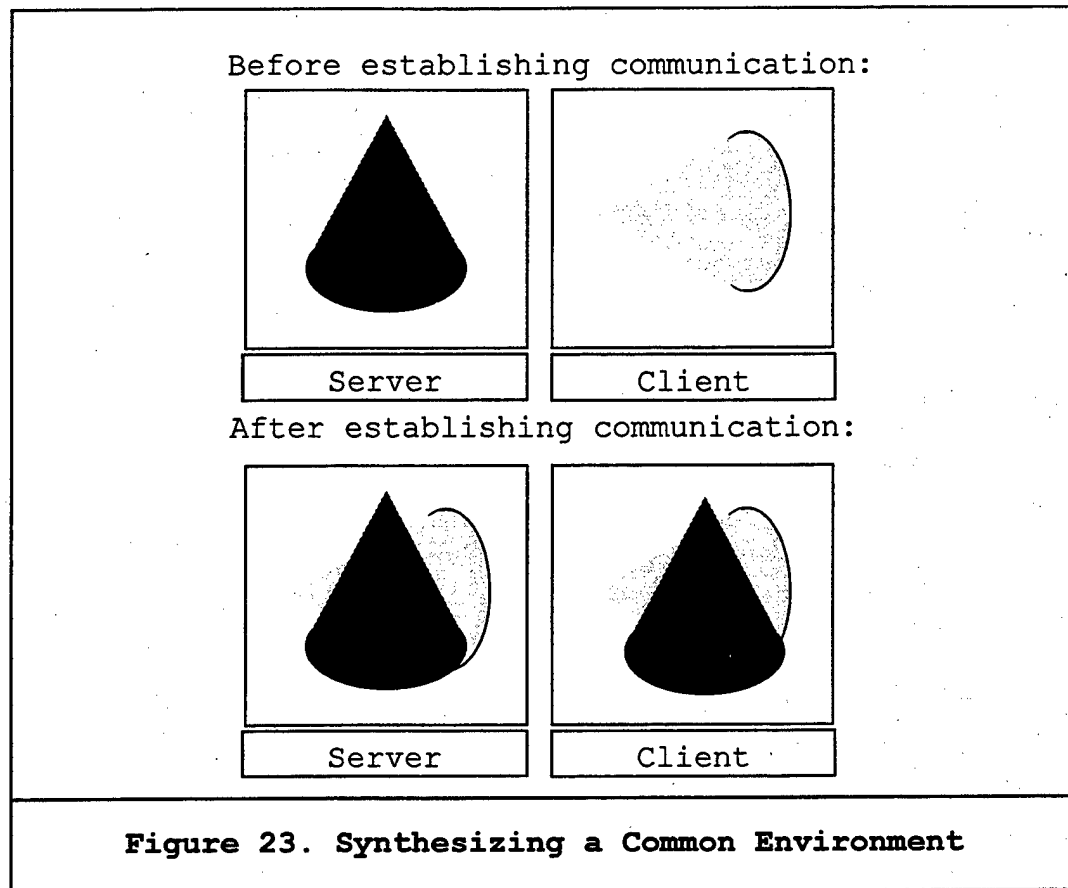
When we were engaged in the SOFT project, a working prototype existed. This prototype consisted of two modules, a server and a client, working together as a pair. Both the client and the server had been implemented in C++ on Silicon Graphics machines running UNIX. This prototype provided a two-way communication between the client and the server (Figure 22). The server and the client were running OpenInventor and rendered a simple cone. After establishing a communication channel, each one (the server and the client) was responsible for delivering its own scene graph

to the other. At this point, both were able to render a synthetic image consisting of each other's cone (Figure 23).



In this prototype, OpenInventor was used as the SSG and reporting mechanism. The ML layer was provided through separate functionality, specifically designed for the OpenInventor implementation. The NSG layer was also separate and provided the above communication functionality. Our objective was to augment the functionality of the first prototype and to design a server module with:

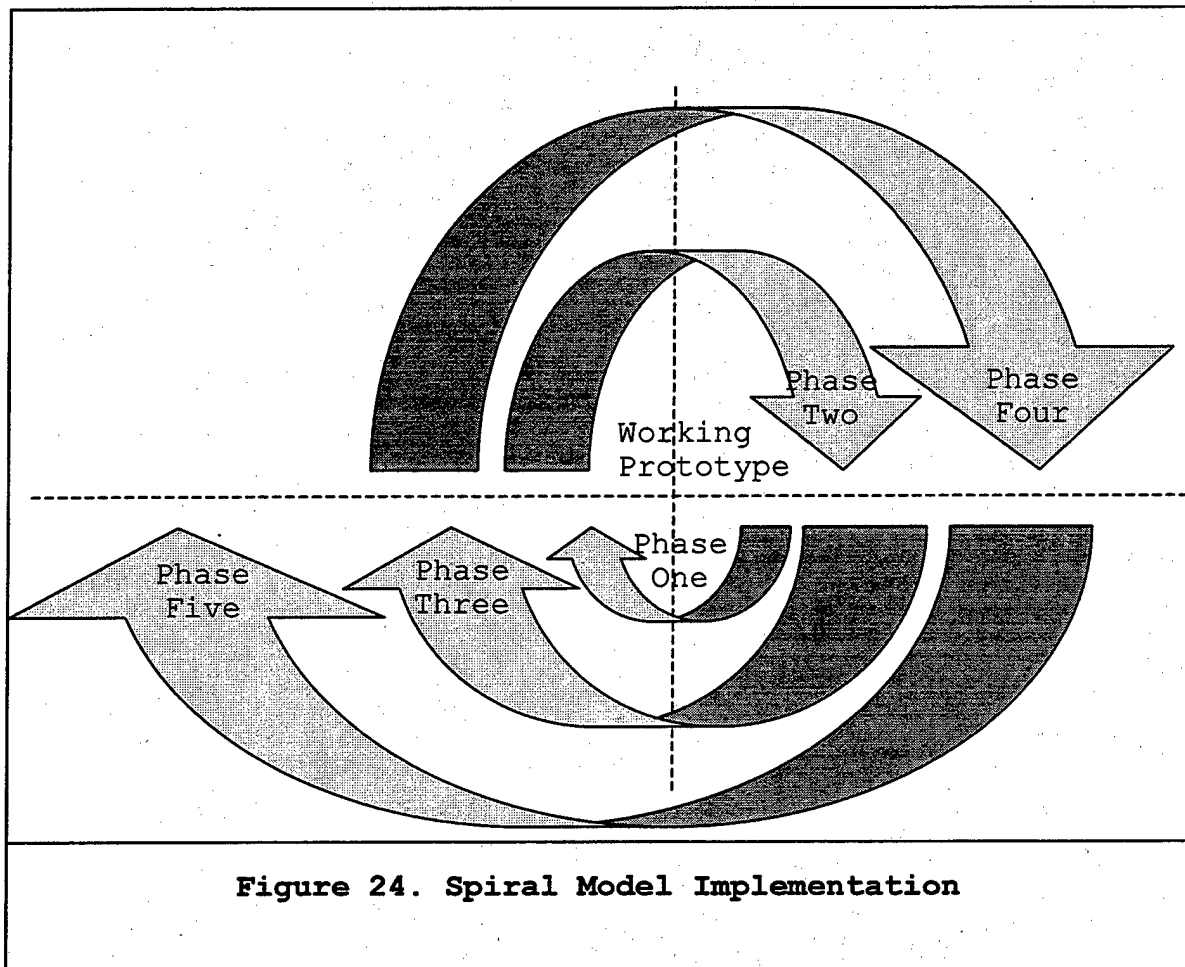
- Multi-Client Functionality;
- Latecomer Support;
- Persistence;
- Locking Mechanism.



According to Schach, some popular software life-cycle models are:

- Build-and-Fix;
- Waterfall;
- Rapid Prototyping;
- Incremental;
- Synthesize and Stabilize;
- Spiral;
- Object Oriented. [SCHA99]

Having the above working prototype as a basis and considering timing limitations and involved risk, we decided to use the spiral model. The spiral model corresponds each cycle to a phase. As shown in Figure 24, we used the working prototype as a core, and we gradually progressed to phase five.



C. PHASES OF DEVELOPMENT

We developed the application in five phases. We present in each phase below, our objectives and the derived conclusions that provided us the feedback to continue onto the next one.

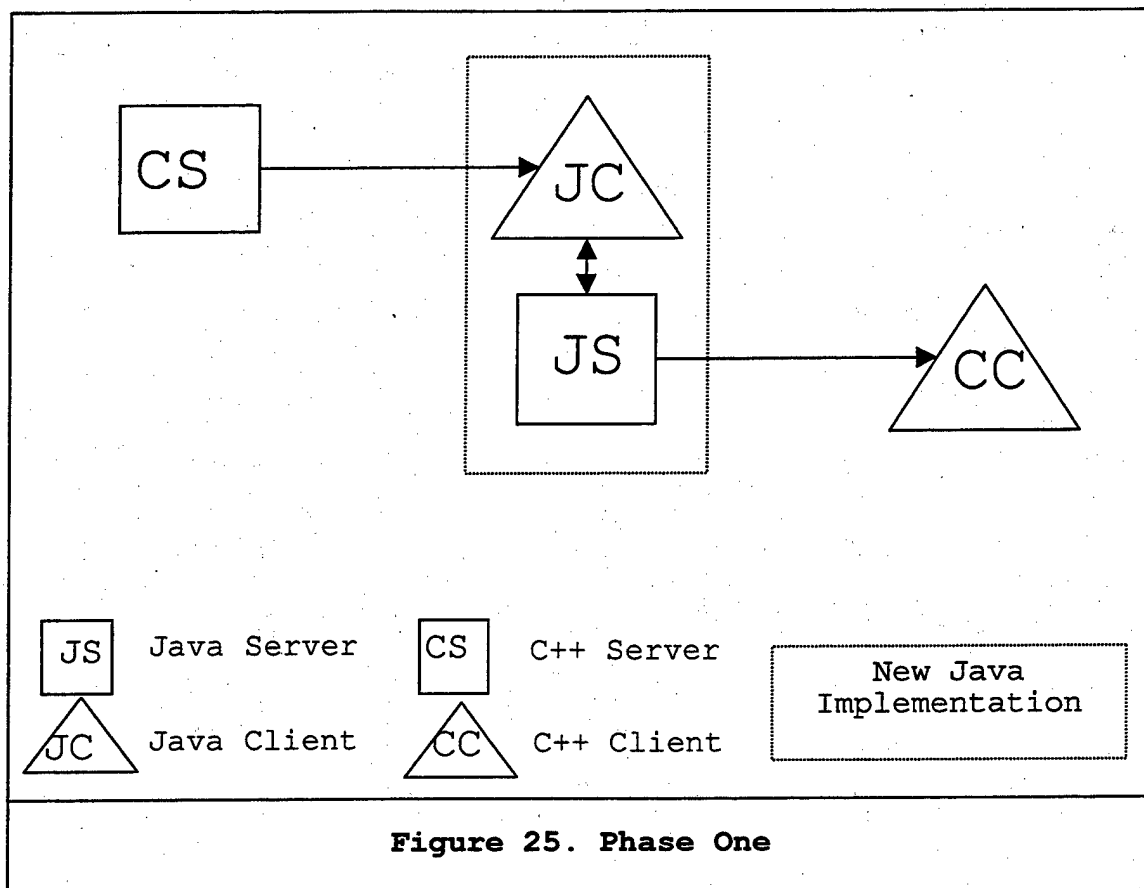
1. Phase One

a) Objective

In order to examine the feasibility of such an independent mechanism, our objective was the intervention of a client/server mechanism between the existing pair of client and server (Figure 25). The benefit from this was a better understanding of the working prototype and the format of exchanged data.

b) Implementation

This intervening mechanism was implemented in Java 1.2 on Windows NT 4.0, as shown in Appendix A. The UML documentation of this phase is shown in Figure 26. The new intervening mechanism was a hybrid client/server system consisting of two modules - the Java client/server module, and the original C++ client/server module. The reason for this was to emulate painlessly the current client/server mechanism.



The server C++ module started the execution of a graphics application. Following this, the Java server and the Java client started. The Java server waited to be invoked by listening to the Java and C++ clients using TCP sockets. Once this communication had been established, the Java server waited for the C++ client requests. Upon request, the Java server opened the communication channel with the original C++ client and sent the data. Then, both of them had a common view of the scene. Whenever the server

or the client made modifications to its own respective scene, these modifications were automatically transferred through the network and reflected to the other.

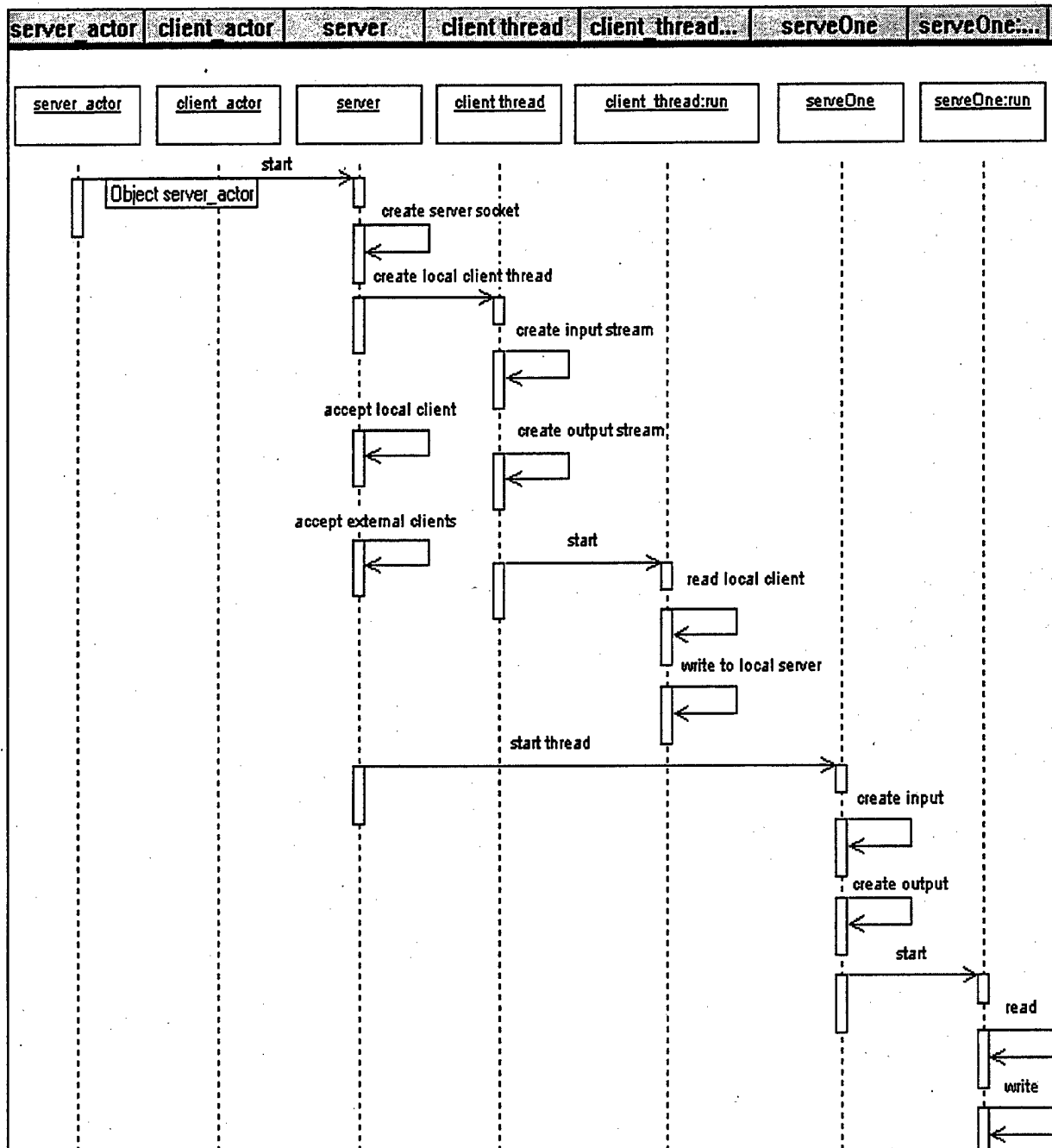


Figure 26. UML Sequence Diagram (Phase One)

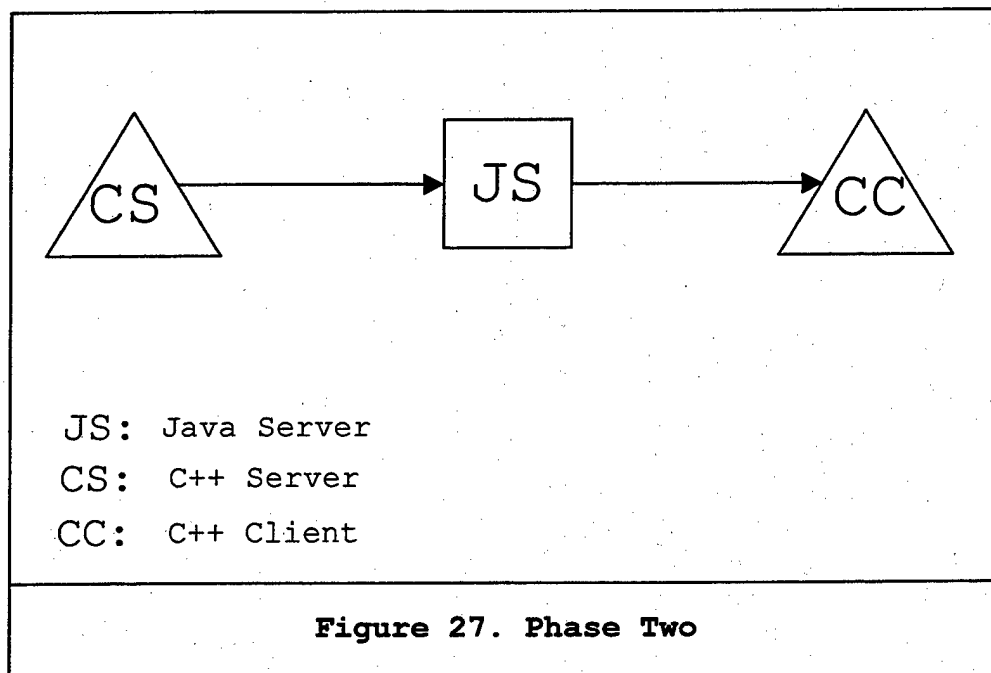
c) Conclusions

In this phase, we introduced seamlessly an independent client/server mechanism in the existing prototype, while maintaining the same functionality. We examined the exchanged data format and were able to build our own server.

2. Phase Two

a) Objective

In this phase, our objective was providing a single-server mechanism between two equivalent existing clients (Figure 27). The benefit of this was forming a new independent server-mechanism. This mechanism was the basis for the new implementation.



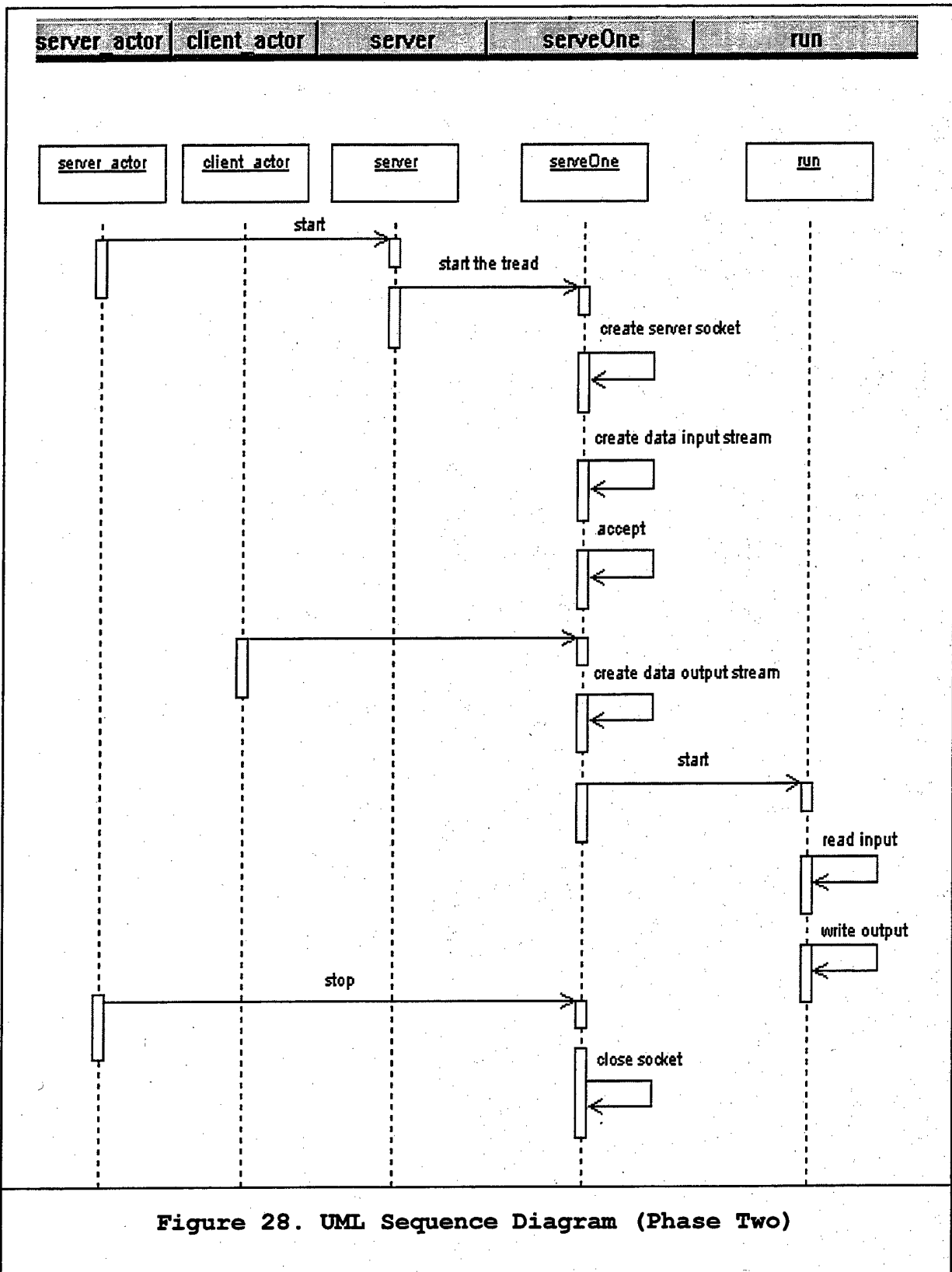
b) Implementation

We improved the phase one prototype, as in Appendix B. The UML documentation is shown in Figure 28. Then, we created the "ServeOne" class as a thread responsible for opening a connection between two predefined C++ clients. Once the connection was established, the thread was reading the input stream from the CS client and was writing the output stream to the CC client. We also created the Server class, responsible for initiating the ServeOne thread.

The CS initiated a new session. Then the JS connected to CS and waited for the CC client to connect. Upon the CC connection, the uni-directional flow of information from the C++ server (CS) to C++ client (CC).

c) Conclusions

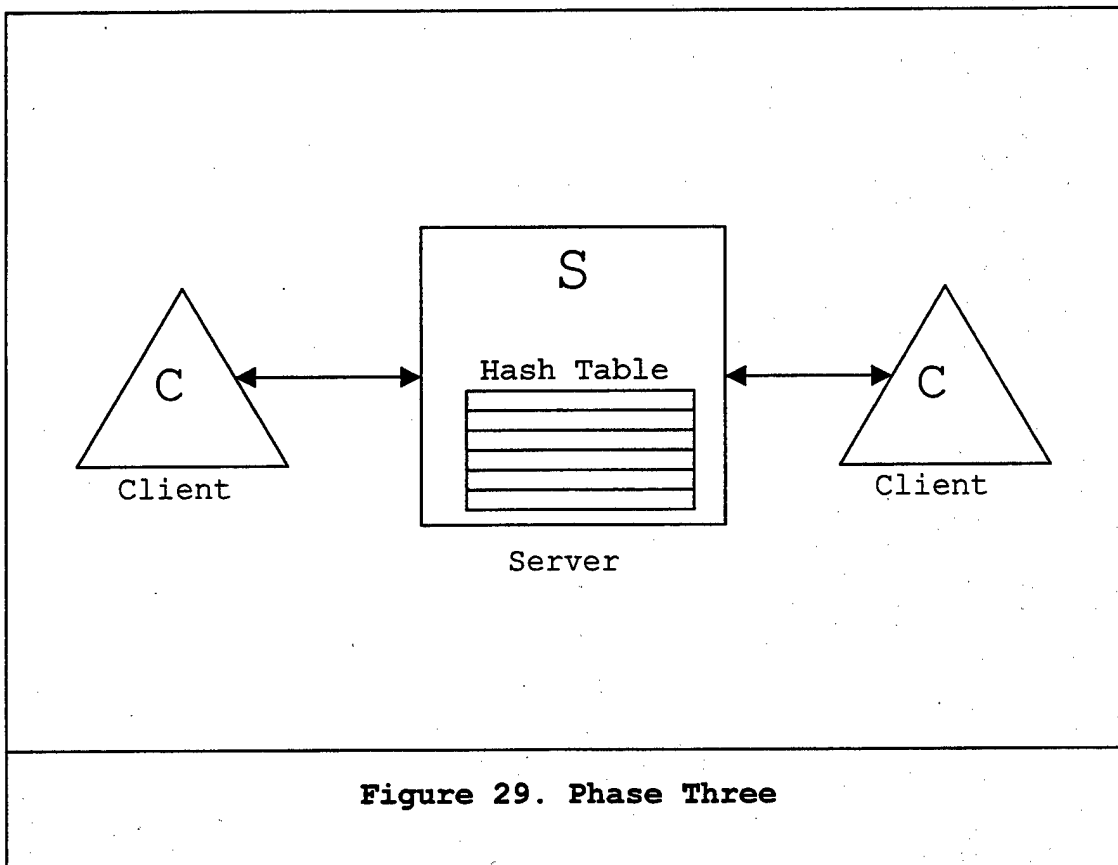
We successfully excluded the Java client implementation from the intermediate server module of Phase One, managing to maintain the same functionality. This simple server module provided scalability for the next phase.



3. Phase Three

a) Objectives

Our objectives were to establish a bi-directional communication and to provide a storing/retrieving mechanism for the NSG messages communicated via the server (Figure 29). This storing/retrieving mechanism demanded a deserialization of the NSG messages. Thus, we would be able to gain scalability for the next phases. Additionally, we would be able to examine the effectiveness of such a mechanism because the existing client implementation required the message retrieval in a specific order.



b) Implementation

First, we modified the server in order to interact with only CC clients. The server listened to CC clients to connect. Upon their connection, the server initiated two threads, one reading the input stream of the first client and writing to the output stream of the second, and vice versa for the second thread.

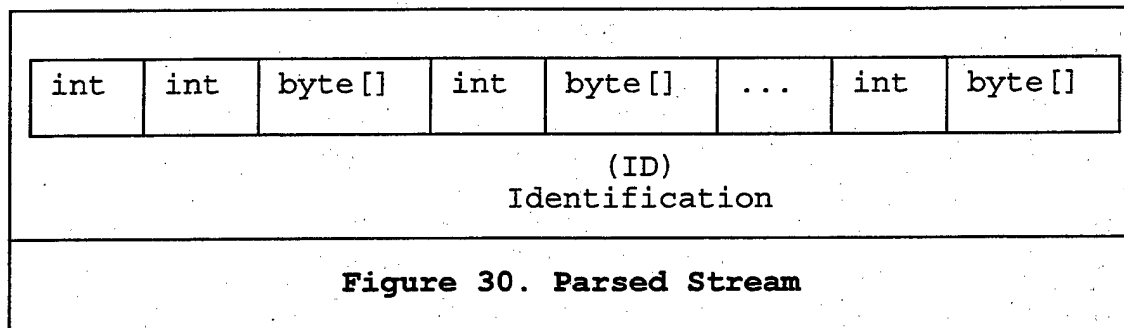
Second, we provided the server with a hash table implementation as the storing data structure.

```
static Hashtable htable = new Hashtable();
```

Moreover, we implemented the method:

```
public void parseStream(byte[] buf, int len)
```

as a deserialization mechanism in order to get the unique identification (ID) of each received message and store it to the hash table (Figure 30).



We chose the Hashtable Java implementation because it guarantees $O(\log(n))$ time cost for the get, put, and remove operations. Moreover, the Hashtable is synchronized [WEB16]. According to this unique ID, the message was either inserted into the hash table (if the ID were a new one) or modified an existing one (if the ID already existed in the table.) Appendix C contains complete implementation. The UML documentation is shown in Figure 31.

c) Conclusions

We managed to establish the bi-directional communication between two clients. Additionally, we stored/retrieved the exchanged data in the hash table. In this manner we maintained a common-shared-view among the clients. As this shared view was stored in the hash table, we were ready to provide latecomer support in the future implementation.

During the testing, we discovered that having stored the messages in the hash table, we were unable to retrieve them in the specific order required by the existing C++ client implementation. Consequently, we concluded that in the next phase we had to change the hash table data structure used for storing and replace it with a *dictionary*.

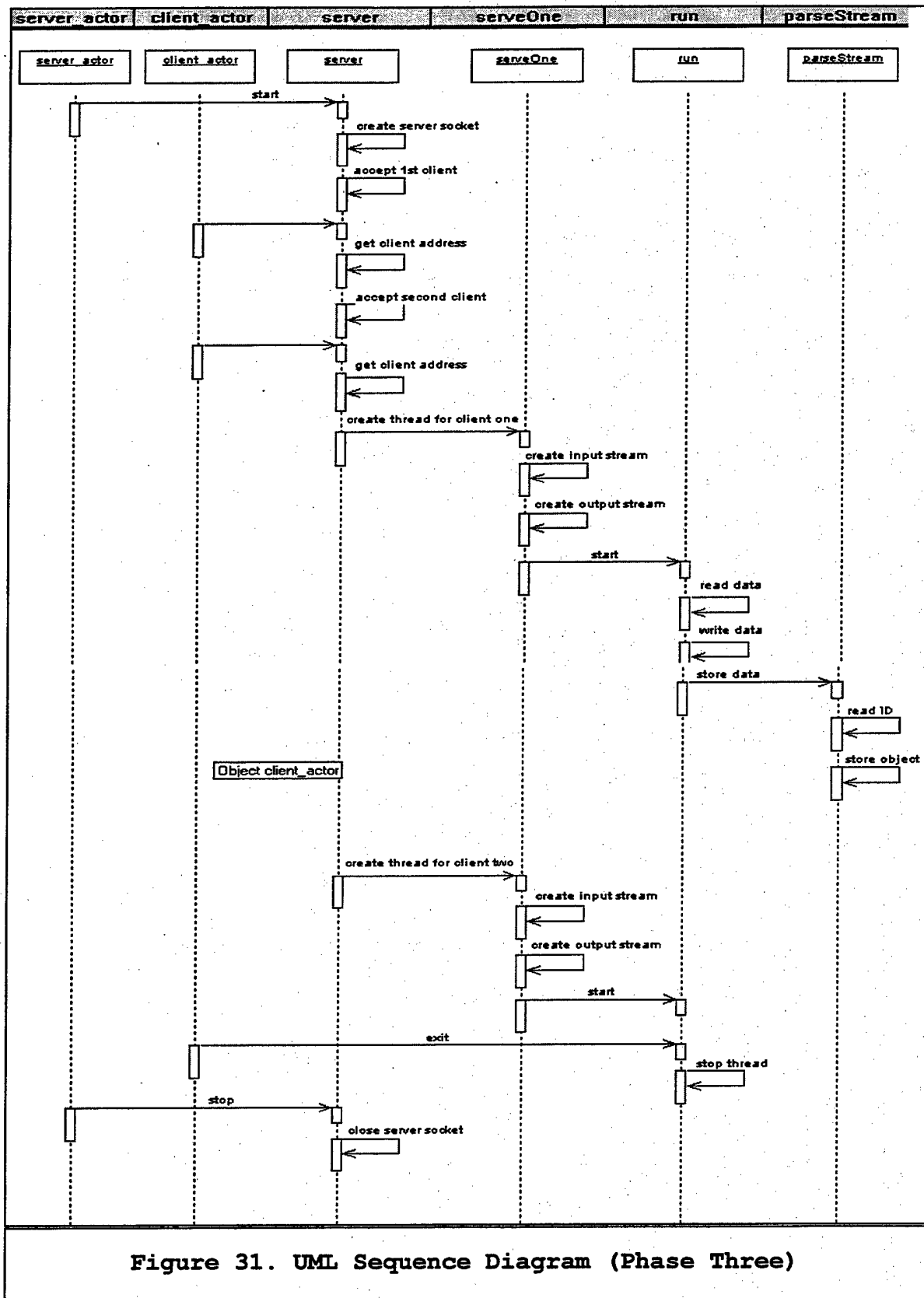
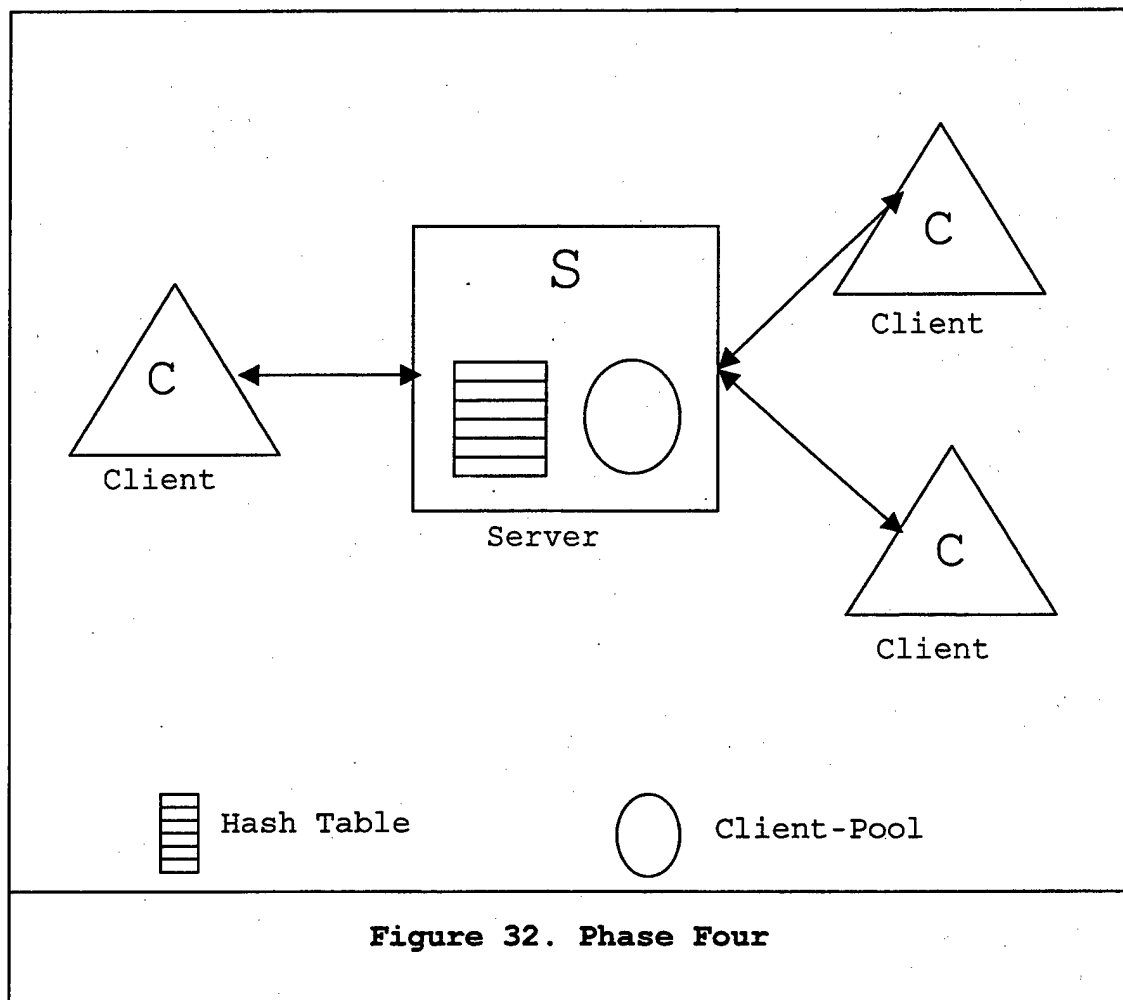


Figure 31. UML Sequence Diagram (Phase Three)

4. Phase Four

a) Objective

Our aim was to provide multi-client functionality to the server (Figure 32.) The benefit from this was latecomer support and persistence. Latecomers would be able to retrieve the common environment from the storing mechanism. Since the hash table proved to be inefficient during Phase Three, we had to replace it with a more appropriate one.



b) Implementation

First, we added a client-pool

```
static Vector castTable = new Vector();
```

to store the currently running clients. In the castTable, we stored the DataOutputStream of each client. Thus, every new message could be retransmitted to all the clients in the pool.

Second, we provided latecomer support, where each new client received all the stored messages. Following this, we replaced the storage data structure (hash table) with a dictionary.

```
public class NSGRepository extends Dictionary
```

Since the dictionary is an abstract class, we decided to implement it using vector. Despite its time cost of $O(n)$ for get, put, and remove, we gained the required functionality of retrieving the records in the order we needed them. This was a temporary solution, as the clients would not require an ordered set of messages in the future.

Third, we provided multi-client support. The server entered in an endless loop where it accepted every new

client, creating a new thread to support it. The new thread was responsible for up-loading the existing environment to the client so that each shared a common view. After this, the new thread started to read from the InputStream of the client and to write to every OutputStream existing in the client pool. The complete implementation is presented in Appendix D. The UML documentation is shown in Figure 33.

c) Conclusions

We managed to store the exchanged messages in a dictionary storage data structure. Consequently, we provided latecomer support and persistence. Furthermore, we were able to function in a multi-client environment.

On the other hand, we concluded that the parsing stream mechanism was able to process only a limited set of message types (e.g. primitive shapes and transformations.) Therefore, we identified the need for an independent mechanism able to read formatted messages rather than simple byte arrays requiring deserialization.

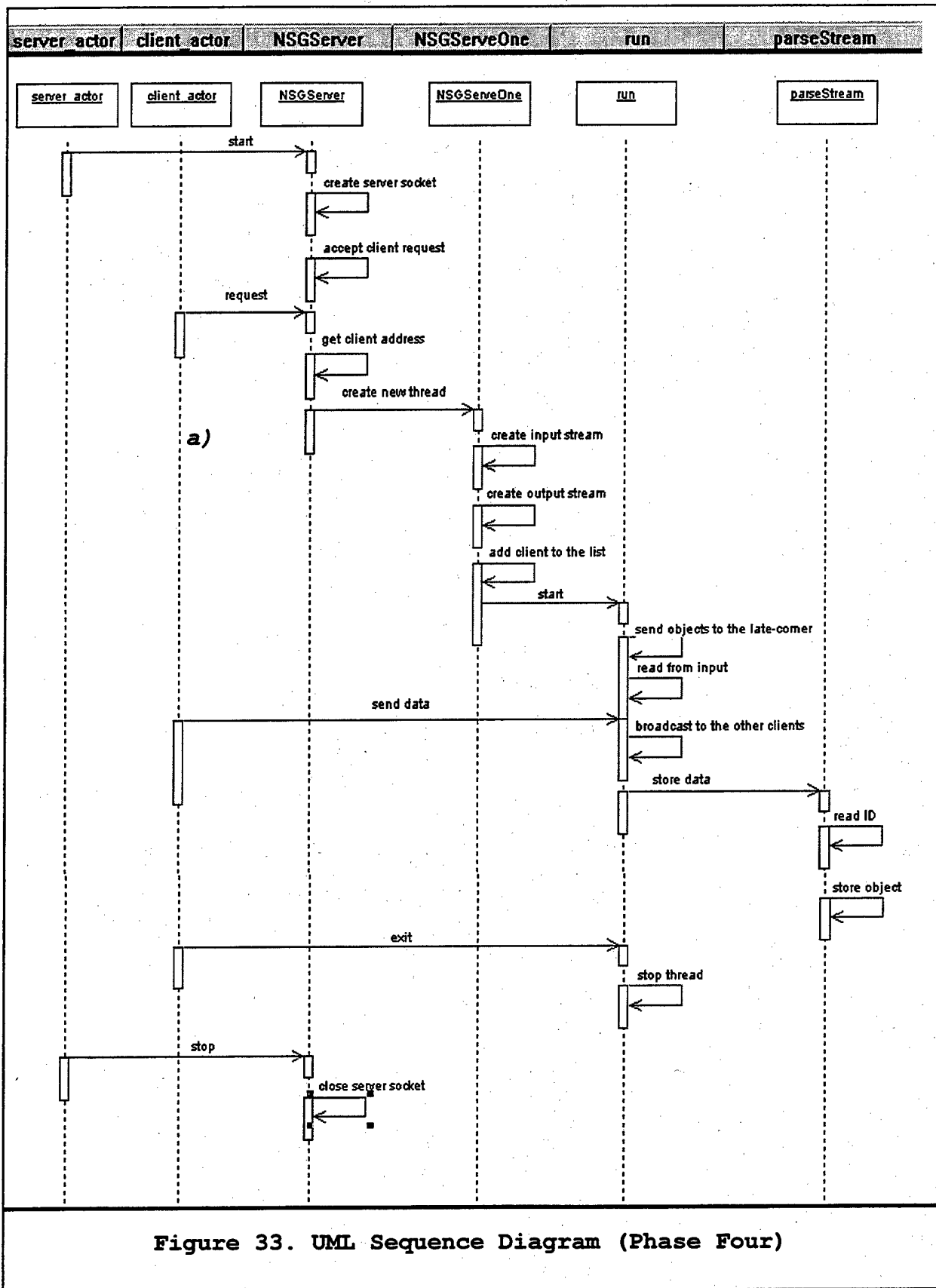


Figure 33. UML Sequence Diagram (Phase Four)

5. Phase Five

a) Objective

Our goal was to provide the ability to read formatted data from the InputStream of each client. The idea was that the server should be unaware of the type of transmitted messages but still be able to dynamically process and store them. This would give better scalability and would make the server implementation less proprietary. As a result, we examined the feasibility of using DaBP.

b) Implementation

Since DaBP requires the use of XML files for reading the format of the protocol, we created a sample XML file as in Appendix E responsible for the representation of the transformation message. The transmitted messages are called Abstract Data Units (ADUs), and they are structured according to the protocol.

We also created two sample Java clients (with no graphical interface) because the C++ clients were not mature enough to be invoked in a reliable testing procedure. The Java implementation of these clients can be found in Appendix F. These Java clients simulated the real clients by sending and receiving messages using DaBP.

The server was responsible for creating the protocol:

```
ProtocolDescription protocol;
```

```
protocol = new ProtocolDescription("SOFTexp.xml");
```

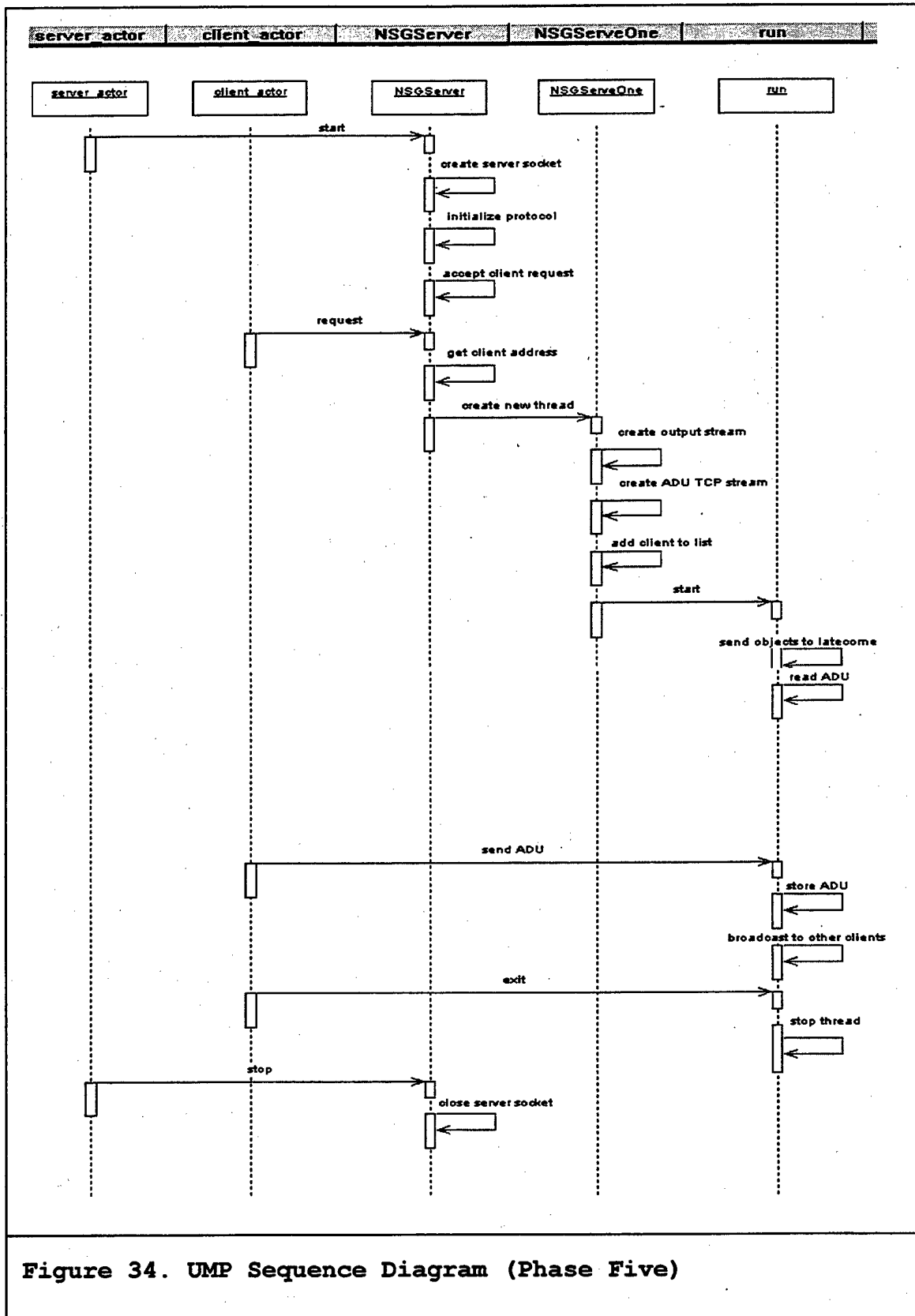
and each thread was responsible for establishing a stream for reading using the ADUStreamTCP class, which reads one or more ADUs transmitted across the TCP socket.

```
ADUStreamTCP tcpStream;
```

```
Socket in_socket;
```

```
tcpStream = new ADUStreamTCP(in_socket, protocol);
```

Next, the thread was able to read from this stream using an ADUData class. This class represents data received from the network in the protocol defined format. Appendix G contains the complete implementation. The UML documentation is shown in Figure 34.



c) Conclusion

After examining the feasibility of using the DaBP, we conclude that:

- DaBP integrated and cooperated efficiently under the current architecture;
- DaBP provided the necessary flexibility for the SOFT project because instead of a stream parsing mechanism, we could directly access the fields of the ADUs;
- DaBP uses XML, which is derived from the ISO Standard Generalized Markup Language (SGML), cooperates with Java, and provides scalability for future implementation;
- DaBP is worth being evolved and included in the SOFT architecture.

VI. CLIENT SIDE ARCHITECTURE AND DESIGN

A. NETWORK SCENE GRAPH (NSG) PROTOCOL

The NSG protocol is an information exchange mechanism between the network layer and the ML. This information is the scene graph appropriately modified and serialized in order to provide platform independence across the network. Each NSG is owned by one host at a time. Specifically, it is composed of nodes and fields. A node consists of:

1. UID

The UID is a unique identifier for the node, consisting of a combination of owner ID and time of creation. A UID is required because we may have multiple clients running on the same host.

2. NSGtag

This defines the type of the node. Different nodes with the same tag may have different type signatures. Once the node has been created, the type signature is not changed. Setting NSGtag only during node creation enforces this policy.

SOFT provides a base set of tags (Table 3) which every SOFT implementation can interpret. The application and the user may define any new tags they wish, ones which are not implemented yet. The tags of Table 4 have already been

designed. Tags are sent as text strings roughly equivalent to notations in a mark-up language.

Table 3. Implemented NSG Tags

NSGsep_node	Contains an array of "noderef" UIDs. The NSG uses left-right/top-down inheritance (as in OpenInventor) instead of top-down inheritance as used in Java3D and FSG
NSGcolor_node	Contains a NSGfield vec3 [rgb]
NSGxform_node	4x4 matrix
NSGcone_node	A cone centered at (0,0,0) that goes from -1 to +1 in each dimension.
NSGcube_node	A cube centered at (0,0,0) that goes from -1 to +1 in each dimension.
NSGcyl_node	A cylinder centered at (0,0,0) that goes from -1 to +1 in each dimension.
NSGsphere_node	A sphere centered at (0,0,0) that goes from -1 to +1 in each dimension.

Table 4. Designed NSG Tags

Array of Tristrips	The number of tristrips. A tristrip contains two arrays of three vectors [position, normal] and two vectors [texture coordinates] and an int[size]
Material	Material properties such as emissive, specular, etc.
Texture	Texture images [contains a URL to the image]
Light	A light object (enumerated such as, spotlights, point-lights)
Camera	Viewport mapping (aspect ratio, etc.) near, far, position, focal distance, typed orientation
FaceSet	Indexed face set: array of three vectors [points] and array of ints.
Transparency	Contains a double [alpha]

3. Owner

This term identifies uniquely the creator or owner of a node. The owner is stored as an IP address and port for communicating updates.

4. Contention Flags

Each node has a set of independent Boolean flags for handling contention. The flags have default settings in the NSG. Any complex sort of contention control requires dropping to a callback. Certain callbacks, which are used frequently in combination to provide desired behavior, are integrated into the system as flags (Table 2).

5. Fields

A field is one of the static, non-extensible sets of enumerated primitive types listed in Table 5.

Table 5. NSG Fields

NSGfield_double	IEEE 64 bit floating point number
NSGfield_int	32 bit signed integer
NSGfield_string	String of ASCII characters
NSGfield_vec3	Three doubles
NSGfield_matrix	4x4 matrix
NSGfield_noderef	A node UID
NSGfield_ntnoderef	A-non-default-traversed reference to a node, in UID format
A runtime-variable-length array of any of the preceding types	

If an application does not provide a traversal behavior for a given tag, then the behavior is changed to traverse any and all "noderef" fields of the node, according to their order. Consequently, the point of an "ntnoderef" is that it allows a user-defined tag node to reference another node without being traversed by applications that don't understand the tag.

The node UID and the owner are required to solve contention issues arising during modification of a remotely-owned object. The ordered list of types of the fields of the node is called the "type signature" of the node. Different nodes with the same tag may have different type signatures. Once a node has been created, the type of signature cannot be changed. A more detailed description of the NSG node is shown in Figure 35.

B. NSG SERIALIZATION/DESERIALIZATION

All the NSG information is encapsulated in a byte array. This byte array is actually transmitted over the network. It consists of:

- An integer that indicates the length of the byte array;
- The body of the message, which carries on the same logic.

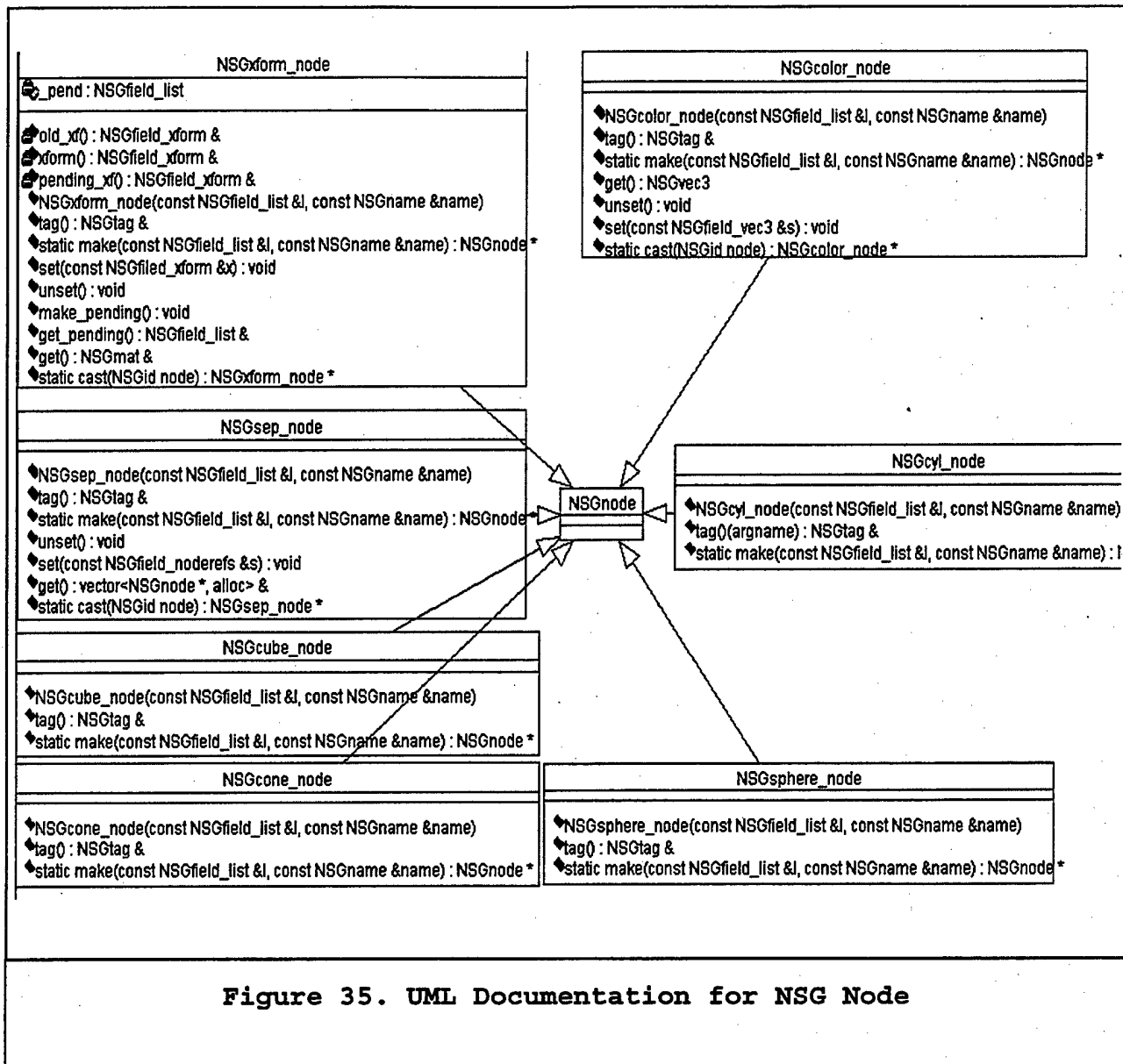


Figure 35. UML Documentation for NSG Node

The serialization is implemented overloading the "<<" operator. According to Meyers, the purpose of operator overloading is that it is easy to read, write, and understand [MEYE97]. Likewise, deserialization is implemented overloading the ">>" operator. During the

deserialization process, the byte array is broken into its individual parts. This architecture provides scalability because the implementers can seamlessly add new data structures.

VII. EMPIRICAL TESTING

A. INTRODUCTION

A shared virtual environment with many clients requires low communications latency. We examined the overhead of our centralized server implementation in a uni-directional data exchange between two clients, both with and without the DaBP.

For experimental purposes, we implemented two specific clients: a sending client (Sclient), and a receiving client (Rclient). A sample packet was used to simulate the exchange of a generic transformation between the two clients. When DaBP was used, the packet was created from the XML file. Without DaBP, the packet was a sample packet taken from the transmission of a real SOFT client. The clients and server ran on an Intel-based computer with Windows NT4.0. The client implementation can be found in Appendix H.

B. TESTING SERVER OVERHEAD WITH DABP

In this experiment we tested the server's overhead involving the DaBP. We transmitted 1,000, 5,000, 10,000, 50,000, 100,000, and 120,000 packets. The results are shown in Table 6. According to the results, as the number of

packets increases the timing cost per packet increases from 1.035 ms to 6.88 ms.

Table 6. Average Server Overhead with DaBP

Packets Transmitted	Timing Cost/ packet
1000	1.035
5000	1.106
10000	1.319
50000	3.085
100000	5.574
120000	6.880

C. TESTING SERVER OVERHEAD WITHOUT DABP

In this experiment we tested the server's overhead without DaBP. The results from the experiment are shown in Table 7. The results reveal a timing cost fluctuating between 6.023 ms and 6.162 ms per packet. The server's overhead is independent from the number of packets transmitted.

Table 7. Average Server Overhead without DaBP

Packets Transmitted	Timing Cost/packet
1000	6.082
5000	6.122
10000	6.049
50000	6.063
100000	6.023
120000	6.162

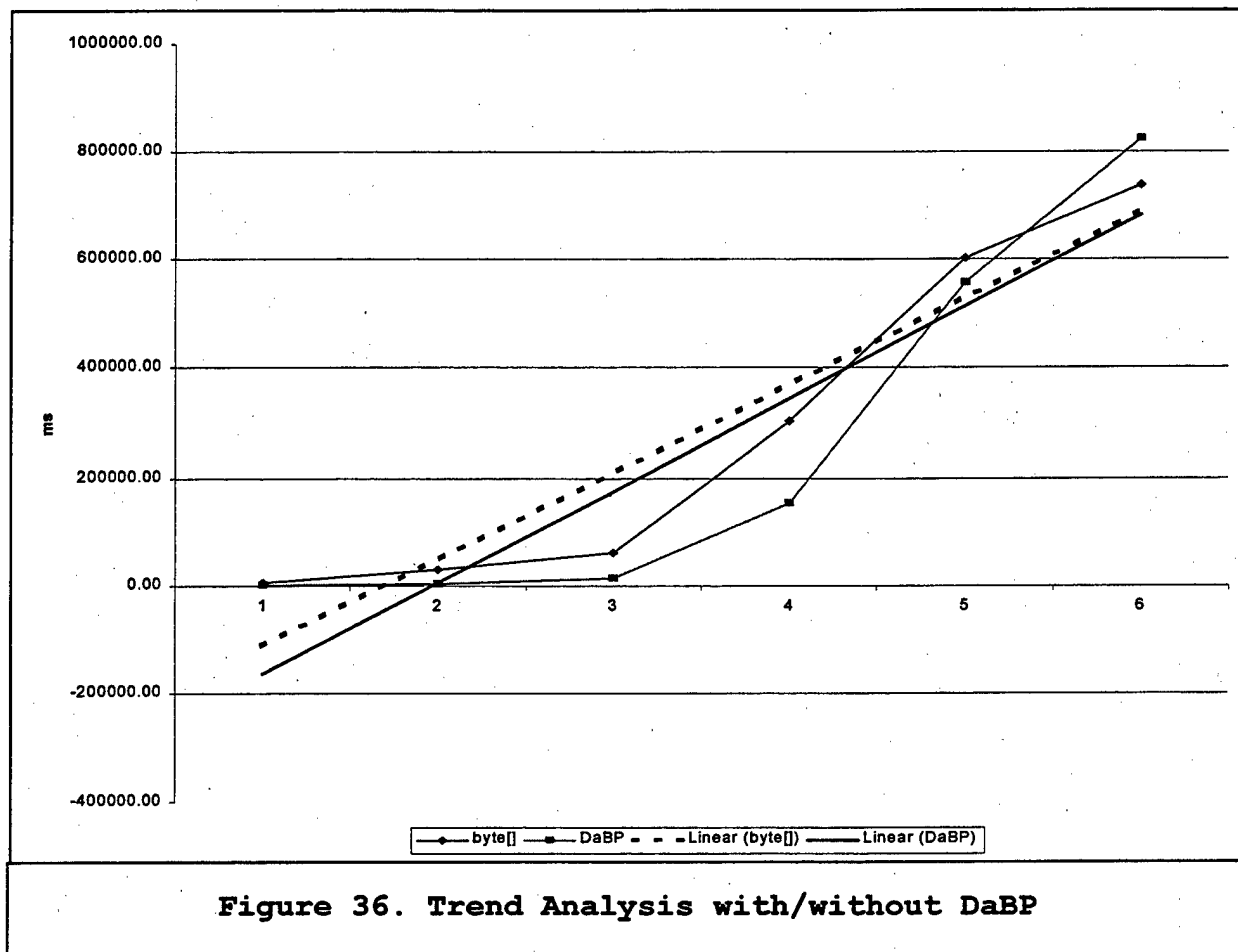
D. PROBLEMS TESTING WITH DABP

During the experimental session, we encountered some problems using DaBP. First, we were unable to install it properly due to hardcoded absolute paths. Thus, we inserted our own absolute paths to make it work. Second, the *String* data type was not supported. We tried to substitute *String* with arrays of unsigned bytes, without success. In order to solve this problem, we created fixed-length byte arrays. Third, during packet transmission null pointer exceptions were thrown. Our first assumption was a buffer overflow, which led us to increase the size of the input and output buffers without result. The problem was solved by replacing

the *InputStream* with *DataInputStream* and the *read* method with *readFully*. Finally we discovered a memory-leak problem, which significantly reduced its performance for large number of packets (over 50,000).

E. CONCLUSIONS

Performing a trend analysis (Figure 36) we discovered that using DaBP gives significant performance enhancement (approximately six time speedup) when the number of transmitted packets is small (less than 50000). When the number of packets increases, the server's overhead is increased, due to memory-leak problems of the DaBP. The parsing of byte streams is a much more time consuming method than extracting the necessary information of a previously formatted-with-DaBP message, but it has a neutral behavior to the number of packets transmitted. The previously discussed problems do not remove DaBP as a viable candidate for integration into the SOFT architecture. But, it needs further implementation and extensive debugging in order to become a stable product.



THIS PAGE WAS INTENTIONALLY LEFT BLANK

VIII. CONCLUSION AND FUTURE WORK

A. INTRODUCTION

This thesis designs and implements a network architecture for distribution of generic scene graphs. We concentrated on the design and implementation of a centralized server that supports multiple clients, providing them with a common shared environment. The results are intended to be incorporated into the SOFT project.

B. CONCLUSIONS

We built a centralized server using Java that provided reliability, persistence, scalability, and latecomer support in a multi-client SOFT environment. This server provides interoperability and can support any SSGs on any platform. We concluded that this implementation demonstrated the benefits of the DaBP.

Empirically testing the server overhead, we discovered that using DaBP reduced the overhead by a factor of six for less than 50,000 packets. The performance and flexibility of DaBP indicates that it is worth the effort to extend it to a stable product and incorporate it into the SOFT architecture. Of course, empirical testing to confirm

performance improvement for large number of packets must be conducted.

C. FUTURE RESEARCH IDEAS

The goal of the SOFT project is network collaboration between computer graphics applications. These applications can be implemented in various languages over different platforms. SOFT clients are currently implemented in C++ on UNIX platforms. They use OpenInventor as the Standard Scene Graph (SSG). In order to fulfill the objectives of the SOFT architecture, the implementation must be extended to include other platforms (e.g., Intel-based), languages (e.g., Java), and SSGs (e.g., Java3D).

As stated in Chapter IV, the SOFT server can be part of a topologically larger network environment. In this environment, various clusters can have their own server modules. Furthermore, the clients have been implemented separately from the server. Merging the server and the client modules would be ideal. This new module would act either as a server and/or as a client. This would mean the module could take turns being a server, and whenever the server module left, it would migrate and transfer duties.

The current implementation of the SOFT client and server modules requires an ordered procedure. Specific

programs must be invoked and executed using command line methods. A user interface could be built in order to act as a SOFT session manager.

Currently, the DaBP library is limited. For instance, a limited set of data types is supported (e.g., integer, float.) Additionally, the libraries are prototype versions and contain errors. In the future, DaBP libraries could be improved and completed resulting in a mature product. Since DaBP provides flexibility, it can be extended to include designs of application-specific protocols that can be changed based on network load.

The feasibility of using multicasting technology for the distribution of generic scene graphs should be examined as multicast hardware becomes more widely available. Also, the use of multi-tier architecture, like CORBA and COM, can be exploited under the SOFT architecture.

D. SUMMARY

The SOFT project approaches networking collaborative virtual environments these environments with the use of the scene graph as bus metaphor. This networking has been implemented with centralized servers responsible for the distribution of the scene graphs. This shared visual environment is the first stepping stone towards networking

client communities. Later goals of SOFT include sharing of entity behaviors and actions, and will be addressed in follow-on work.

APPENDIX A - JAVA CODE FOR "PHASE ONE"

```
//-----  
// Filename: Server.java  
// Date: 20-April-99  
// Compiler: JDK 1.2  
//-----  
import java.net.*;  
import java.io.*;  
  
/**  
 * Server for "Phase One". Creates a client/server mechanism between  
 * the existing pair of SOFT client and server.  
 *  
 * @author P.Fiabolis,G.Prokopakis  
 */  
public class Server {  
  
    /*  
     * Main function  
     *  
     * @param args: Command line arguments  
     * @exception IOException  
     */  
    public static void main (String[] args) throws IOException {  
  
        ServerSocket s = new ServerSocket(9999);  
        ServerSocket is = new ServerSocket(8099);  
        System.out.println("Server started: " + s);  
        System.out.println("Local server started: " + is);  
  
        InetAddress addr = InetAddress.getByName("royal");  
        new ClientThread(addr);  
        System.out.println("Connection with master started ...");  
  
        Socket in_socket = is.accept();  
  
        try {  
            while(true) {  
                Socket socket = s.accept();  
                try {  
                    System.out.println("Slave found ...");  
                    new ServeOne(socket, in_socket);  
                }  
                catch(IOException e) {  
                    System.out.println("Server.main: IOException");  
                    socket.close();  
                }  
            } // end while  
        }  
        finally {  
            s.close();  
            is.close();  
        }  
    }  
}
```



```

    } // end of main()
} // end of class Server
// end of file Server.java

```

```

//-----
// Filename: ServeOne.java
// Date: 20-April-99
// Compiler: JDK 1.2
//-----
import java.io.*;
import java.net.*;

/**
 * Thread running for each connection. A simple client/server
 * mechanism, reading from the input stream and writing to
 * the output stream.
 *
 * @author P.Fiabolis,G.Prokopakis
 */
class ServeOne extends Thread {

    /**
     * Socket of the current thread.
     */
    private Socket socket;

    /**
     * Stream to read from.
     */
    DataInputStream in;

    /**
     * Output stream.
     */
    DataOutputStream out;

```

```

/*
 * Class constructor
 *
 * @param s: Socket for output
 * @param in_socket: Socket for input
 * @exception IOException
 */
public ServeOne(Socket s, Socket in_socket) throws IOException {
    socket = s;

    in = new DataInputStream(in_socket.getInputStream());
    out = new DataOutputStream(s.getOutputStream());

    start();
} // end of ServeOne

/*
 * Thread main loop. Performs the read/write operation until
 * the clients are disconnected.
 */
public void run() {
    byte[] buf = new byte[1024];
    int len = 0, tmp = 0;

    try {
        while(true) {
            try {
                len = in.readInt();
                in.readFully(buf, 0, len);

                out.writeInt(len);
                out.write(buf, 0, len);
            }
            catch(IOException e) {
                System.out.println("Cannot read ...");
            }
        }
    }
    finally {
        // Always close it:
        try {
            socket.close();
        }
        catch (IOException e) {
            System.out.println("ServeOne.run: IOException");
        }
    }
} // end of run()

} // end of class ServeOne
// end of file ServeOne.java

```

```

//-----
// Filename: ClientThread.java
// Date: 20-April-99
// Compiler: JDK 1.2
//-----
import java.net.*;
import java.io.*;

/**
 * Thread implementation of the internal client module.
 * This module reads from the C++ SOFT Server, and writes
 * to the Java server module.
 *
 * @author P.Fiabolis,G.Prokopakis
 */
class ClientThread extends Thread {

    /**
     * Socket of the current thread.
     */
    private Socket socket;

    /**
     * Output socket.
     */
    private Socket out_socket;

    /**
     * Stream to read from.
     */
    DataInputStream data_in;

    /**
     * Output stream.
     */
    DataOutputStream data_out;

    /**
     * Class constructor
     *
     * @param addr: Inet address where this client should connect
     */
    public ClientThread(InetAddress addr) {
        System.out.println("Making client ...");
        try {
            socket = new Socket(addr, 8080);

            InetAddress self = InetAddress.getByName("127.0.0.1");
            out_socket = new Socket(self, 8099);
        }
        catch (IOException e) {
            // If the creation of the socket fails, no need for cleanup.
        }

        try {
            data_in = new DataInputStream(socket.getInputStream());

```

```

        data_out = new
        DataOutputStream(out_socket.getOutputStream());
        start();
    }
    catch (IOException e) {
        // The socket should be closed on any failures
        // other than the socket constructor :
        try {
            socket.close();
            out_socket.close();
        }
        catch (IOException e2) {
        }
    }
}

/*
 * Thread main loop. Performs the read(from C++ server)/write
 * operation until the clients are disconnected.
 */
public void run() {
    byte[] buf = new byte[1024];
    int len = 0, tmp = 0;

    try {
        while(true) {
            try {
                len = data_in.readInt();

                data_in.readFully(buf, 0, len);
                data_out.writeInt(len);
                data_out.write(buf, 0, len);
            }
            catch(IOException e) {
                System.out.println("Cannot read ...");
            }

            } // end while
        }
        finally {
            // Always close it:
            try {
                socket.close();
                out_socket.close();
            }
            catch (IOException e) {
            }
        }
    } // end of run()
} // end of class ClientThread
// end of file ClientThread.java

```


APPENDIX B - JAVA CODE FOR "PHASE TWO"

```
//-----  
// Filename: Server.java  
// Date:      04-May-99  
// Compiler:  JDK 1.2  
//-----  
import java.net.*;  
import java.io.*;  
  
/**  
 * Server for "Phase Two". Performs the read/write operation  
 * between a C++ SOFT server and a C++ SOFT client, without  
 * needing an internal client module.  
 *  
 * @author P.Fiabolis,G.Prokopakis  
 */  
public class Server {  
  
    /*  
     * Main function  
     *  
     * @param args: Command line arguments  
     * @exception IOException  
     */  
    public static void main (String[] args) throws IOException {  
        new ServeOne();  
    } // end of main  
} // end of class Server  
  
// end of file Server.java
```

```
//-----  
// Filename: ServeOne.java  
// Date:      04-May-99  
// Compiler:  JDK 1.2  
//-----  
import java.io.*;  
import java.net.*;  
  
/**  
 * Thread running for each connection. A simple client/server  
 * mechanism, reading from the input stream and writing to  
 * the output stream.  
 *  
 * @author P.Fiabolis,G.Prokopakis  
 */
```

```

    */
    class ServeOne extends Thread {

    /**
     * Sockets for read and write.
     */
    private Socket in_socket, out_socket;

    /**
     * Server Socket for the connection of the C++ client.
     */
    private ServerSocket server_socket;

    /**
     * Stream to read from.
     */
    DataInputStream in;

    /**
     * Output stream.
     */
    DataOutputStream out;

    /**
     * Class constructor
     *
     * @exception IOException
     */
    public ServeOne() throws IOException {
        server_socket = new ServerSocket(8080);
        InetAddress addr = InetAddress.getByName("royal");
        in_socket = new Socket(addr, 9999);

        in = new DataInputStream(in_socket.getInputStream());

        out_socket = server_socket.accept();
        out = new DataOutputStream(out_socket.getOutputStream());

        start();
    }

    /**
     * Thread main loop. Performs the read/write operation until
     * the clients are disconnected.
     */
    public void run() {
        byte[] buf = new byte[1024];
        int len = 0, tmp = 0;

        try {
            while(true) {
                try {
                    len = in.readInt();

```

```

        in.readFully(buf, 0, len);

        out.writeInt(len);
        out.write(buf, 0, len);
    }
    catch(IOException e) {
        System.out.println("Cannot read ...");
    }
}

}
finally {
    // Always close it:
    try {
        in_socket.close();
        out_socket.close();
        server_socket.close();
    }
    catch (IOException e) {
    }
}
} // end of run()

} // end of class ServeOne

// end of file ServeOne.java

```


APPENDIX C - JAVA CODE FOR "PHASE THREE"

```
//-----  
// Filename: Server.java  
// Date:      03-June-99  
// Compiler:  JDK 1.2  
//-----  
import java.net.*;  
import java.io.*;  
  
/**  
 * Server for "Phase Three". Performs the read/write operation  
 * between two C++ SOFT clients.  
 *  
 * @author P.Fiabolis,G.Prokopakis  
 */  
public class Server {  
  
    /*  
     * Main function  
     *  
     * @param args: Command line arguments  
     * @exception IOException  
     */  
    public static void main (String[] args) throws IOException {  
  
        if(args.length != 3) {  
            usageMessage();  
            System.exit(0);  
        }  
  
        String server_name = args[0];  
  
        ServerSocket server_socket;  
        Socket in_socket, out_socket;  
        InetAddress addr;  
  
        try {  
            server_socket = new ServerSocket(Integer.parseInt(args[2]));  
  
            in_socket = server_socket.accept();  
            InetAddress client_addr1 = in_socket.getInetAddress();  
  
            System.out.println("Client found ..." +  
client_addr1.getHostName());  
  
            out_socket = server_socket.accept();  
            InetAddress client_addr2 = out_socket.getInetAddress();  
  
            System.out.println("Client found ..." +  
client_addr2.getHostName());  
  
            try {  
                new ServeOne(in_socket, out_socket, "SOFT client 1");  
            }  
        }  
    }  
}
```

```

        new ServeOne(out_socket, in_socket, "SOFT client 2");

        System.out.println(Thread.activeCount());
        while(Thread.activeCount() > 1) {
            } // end while
        }
        finally {
            try {
                in_socket.close();
                out_socket.close();
            }
            catch (IOException e) {
                System.out.println("Could not close socket.");
            }
        }
    }
    catch(IOException ie) {
        System.out.println("Server problem");
        System.exit(0);
    }
} // end of main

/*
 * Provide help message if the command line arguments
 * are not valid.
 */
public static void usageMessage() {
    System.out.print("Usage: java Server <server name>");
    System.out.println(" <server port> <client port>");
    System.out.println();
    System.out.println("Exaple: java Server venus 9000 8000");
    System.out.println("\tvenus - machine running the SOFT server");
    System.out.println("\t9000 - venus machine port");
    System.out.println("\t8000 - client machine port");
} // end of usageMessage

} // end of class Server

// end of file Server.java

```

```

//-----
// Filename: NSGStreamRecord.java
// Date:    04-May-99
// Compiler: JDK 1.2
//-----

/**
 * Implementation of a record to be stored by the SOFT server.
 *
 * @author P.Fiabolis,G.Prokopakis
 */
class NSGStreamRecord {

    /**
     * Unique key for the message.
     */
    String key;

    /**
     * Message as it was received from the network.
     */
    byte[] data;

    /**
     * Class constructor
     *
     * @param id: Message ID
     * @param len: Message length
     * @param record: Original message
     */
    public NSGStreamRecord(byte[] id, int len, byte[] record) {
        key = new String(id, 0, len);
        data = new byte[65535];
        data = record;
    } // end of NSGStreamRecord

} // end of class NSGStreamRecord

// end of file NSGStreamRecord.java

```


APPENDIX D - JAVA CODE FOR "PHASE FOUR"

```
//-----  
// Filename: NSGServer.java  
// Date: 18-June-99  
// Compiler: JDK 1.2  
//-----  
import java.net.*;  
import java.io.*;  
  
/**  
 * Main class for the NSG server. Enters an endless loop waiting  
 * for client requests.  
 *  
 * @author P.Fiabolis,G.Prokopakis  
 */  
  
public class NSGServer {  
  
    /*  
     * App main function.  
     *  
     * @param args: Command line param. Should have the port number.  
     * @exception IOException  
     */  
    public static void main (String[] args) throws IOException {  
        if(args.length != 1) {  
            usageMessage();  
            System.exit(0);  
        }  
  
        ServerSocket server_socket;  
        InetAddress addr;  
  
        int counter = 1;  
  
        try {  
            server_socket = new ServerSocket(Integer.parseInt(args[0]));  
            while(true) {  
                Socket in_socket = server_socket.accept();  
  
                addr = in_socket.getInetAddress();  
  
                System.out.println("Welcome " + addr.getHostName());  
  
                try {  
                    new NSGServeOne(in_socket, counter++);  
                }  
                catch(IOException e) {  
                    in_socket.close();  
                }  
            } // end while  
        }  
    }  
}
```

```

        catch(IOException e) {
            System.out.println("Cannot initialize server. Exiting ...");
            System.exit(0);
        }

    } // end of main

    /**
     * Message to display in case of invalid command line arguments.
     */
    public static void usageMessage() {
        System.out.println("Usage : java Server <port number>");
        System.out.println();
        System.out.println("Example: java Server 9999");
    } // end of usageMessage

} // end of class NSGServer

// end of file NSGServer.java

```

```

//-----
// Filename: NSGServeOne.java
// Date: 18-June-99
// Compiler: JDK 1.2
//-----

import java.io.*;
import java.net.*;
import java.util.Hashtable;
import java.util.Enumeration;
import java.util.Vector;

/**
 * An instance of this class is running for every client connected to
 * the
 * NSGServer.
 *
 * @author P.Fiabolis,G.Prokopakis
 */
class NSGServeOne extends Thread {

    /**
     * Unique ID for each thread. For the moment this is a simple counter
     */
    int threadID;

    /**
     * Stream where this thread reads from

```

```

    */
    DataInputStream in;

    /**
     * Stream where this thread will write the NSG's stored in the
     repository.
     */
    DataOutputStream me;

    /**
     * Table holding the ports for all conected clients. Common for all
     threads
     */
    static Vector castTable = new Vector();

    /**
     * Table holding SG data sent from clients. Common for all threads
     */
    static NSGRepository table = new NSGRepository();

    /**
     * Semaphore for thread exit.
     */
    boolean stopThread = false;

    /**
     * Socket associated with this thread.
     */
    Socket threadSocket;

    /*
     * Class constructor
     *
     * @param in_socket: Socket for this thread connection
     * @param id: unique ID for this client. For the moment just a counter
     * @exception IOException
     */
    public NSGServeOne(Socket in_socket, int id) throws IOException {
        threadID = id;

        threadSocket = in_socket;

        in = new DataInputStream (in_socket.getInputStream());
        me = new DataOutputStream(in_socket.getOutputStream());

        // Add the new comer to the client list.
        castTable.addElement(me);

        start();
    } // end of NSGServeOne

    /*

```



```

* Thread main loop. Runs until thread needs to be alive.
*
*/
public void run() {
    int len = 0;
    DataOutputStream out;
    // Send the data (if any) from the repository to the newcomer.
    try {
        Enumeration enum = NSGServeOne.table.elements();
        while(enum.hasMoreElements()) {
            NSGStreamRecord s = (NSGStreamRecord)enum.nextElement();
            if (s.threadID != threadID) {
                System.out.println("TABLE: " + s.key);

                byte[] tmpBuf = new byte[65535];
                tmpBuf = s.data;
                me.writeInt(s.dataLength);
                me.write(tmpBuf, 0, s.dataLength);
                me.flush();
            }
        }
    } catch(IOException e) {
        System.out.println("Thread stopped. Connection with " +
            threadID + " failed.");
        closeThread();
    }
    while(!stopThread) {
        try {
            len = in.readInt();

            byte[] buf = new byte[len];
            in.readFully(buf, 0, len);

            // Send the message to everyone
            Enumeration enum = castTable.elements();
            while(enum.hasMoreElements()) {
                out = (DataOutputStream)enum.nextElement();

                if (out != me) {
                    out.writeInt(len);
                    out.write(buf, 0, len);
                }
            }
        }
    }
}

```

```

        } // end while

        // Store or update the data.
        parseStream(buf, len);
    }
    catch(IOException e) {
        System.out.println("Goodbye " + threadID);
        closeThread();
    }

    } // end while(!stopThread)
} // end of run

/*
 * Close the thread after finishing connection or failure
 */
public void closeThread() {
    try {
        threadSocket.close();
        castTable.removeElement(me);
        stopThread = true;
    }
    catch(IOException e) {
        System.out.println("ERROR: Could not close socket @
closeThread");
    }
} // end of closeThread

/*
 * Parse data received from the network and store them in an appropriate
 * repository. Data for existing objects are been updated.
 *
 * @param buf: Byte array send from the client.
 * @param len: Length of the byte array.
 */
public void parseStream(byte[] buf, int len) {
    byte[] buffer = new byte[65535];

    buffer = buf;

    ByteArrayInputStream byte_stream;

    DataInputStream data_stream;

    int dataLen = len;

    int headerLen; // Size of data header

    int nameLen; // Size of NSG node name

    int keyLen; // Length of unique key for this data

```

```

int lenLeft;

int bytesToRead;

byte[] tmp_buf = new byte[65535];

// Create a new stream just to break it and extract the NSG ID.

byte_stream = new ByteArrayInputStream(buf);
data_stream = new DataInputStream(byte_stream);

try {
    bytesToRead = data_stream.readInt();

    len = len - 4;

    if (bytesToRead == 4) {
        // Read in header

        headerLen = data_stream.readInt();
        len = len - 4;
        data_stream.readFully(tmp_buf, 0, headerLen);
        len = len - headerLen;
        String header = new String(tmp_buf, 0, headerLen);

        // Read in node name
        nameLen = data_stream.readInt();
        len = len - 4;
        data_stream.readFully(tmp_buf, 0, nameLen);

        if (!header.equals("NSGnew")) {
            // This record doesn't describe a NSGnode, so make a
            // key by concatenating the header with the "key" (node
name)
            String name = new String(tmp_buf, 0, nameLen);
            String final_key = header + "|" + name;

            tmp_buf = final_key.getBytes();
            keyLen = final_key.length();
        }
        else {
            keyLen = nameLen;
        }

        NSGStreamRecord stream_rec = new NSGStreamRecord(tmp_buf,
            keyLen, buffer, dataLen, threadID);

        Object val = table.put(stream_rec.key, stream_rec);
        System.out.println(stream_rec.key);

        if (val == null) {
            System.out.println(threadID + ": New record added.");
        }
        else {
            System.out.println(threadID + ": Record found +
updated.");
        }
    }
}

```

```

    }
    else {
        System.out.println("ERROR: No recorded message. Length = " +
            bytesToRead + " @ parseStream.");
    }
}
catch(IOException ie) {
    System.out.println("ERROR: Problem reading integer @
parseStream.");
}
} // end of parseStream
} // end of class NSGServeOne
// end of file NSGServeOne.java

```

```

//-----
// Filename: NSGStreamRecord.java
// Date:      18-June-99
// Compiler:  JDK 1.2
//-----

```

```

/**
 * NSG record. Stored in the NSGRepository.
 * Should be modified to match the appropriate structure of the NSG.
 *
 * @author P.Fiabolis,G.Prokopakis
 */
class NSGStreamRecord {

    /**
     * Unique key of the NSG
     */
    String key;

    /**
     * Byte array. It is stored as it is read from the client.
     */
    byte[] data;

    /**
     * Length of the byte array (data).
     */
    int dataLength;

    /**
     * Thread that this NSG came from. Used in order to avoid retransmitting
     this back.
     */
}

```

```

int threadID;

/*
 * Class constructor.
 *
 * @param recKey: Unique ID of the NSG.
 * @param keyLen: Length of the key.
 * @param record: Byte array as it is read from the client.
 * @param dataLen: Length of the byte array (record)
 * @param id: Thread ID.
 */
public NSGStreamRecord(byte[] recKey, int keyLen, byte[] record, int
dataLen, int id) {

    key = new String(recKey, 0, keyLen);

    data = new byte[65535];

    data = record;

    dataLength = dataLen;

    threadID = id;
} // end of NSGStreamRecord
} // end of class NSGStreamRecord
// end of file NSGStreamRecord.java

```

```

//-----
// Filename: NSGRepository.java
// Date:      18-June-99
// Compiler:  JDK 1.2
//-----
import java.util.*;
import java.io.*;

/**
 * Implementation of the data structure where NSG's are stored.
 * For the moment a Vector is used in order to provide NSG's
 * to the new comers in the order they have arrived.
 * This is going to be replaced by the TreeMap class (jdk 1.2).
 *
 * @author P.Fiabolis,G.Prokopakis
 */
public class NSGRepository extends Dictionary {

```

```

/**
 * Vector to store the keys of the NSG's.
 */
private Vector Keys = new Vector();

/**
 * Vector to store the byte arrays (data).
 */
private Vector Values = new Vector();

/**
 * Gives the number of the contents of the repository.
 */
public int size() {
    return Keys.size();
}

/**
 * Check if the vector is empty.
 */
public boolean isEmpty() {
    return Keys.isEmpty();
}

/**
 * Insert a new NSG record in the repository.
 *
 * @param key: Unique ID of the NSG.
 * @param value: NSG record to be stored.
 */
public Object put(Object key, Object value) {
    if(Keys.contains(key)) {
        int index = Keys.indexOf(key);
        Values.setElementAt(value, index);
        return key;
    }
    else {
        Keys.addElement(key);
        Values.addElement(value);
        return null;
    }
}

/**
 * Get the record of the NSG with ID = key.
 *
 * @param key: Unique ID of NSG to look for.
 */
public Object get(Object key) {
    int index = Keys.indexOf(key);

```

```

        if(index == -1) {
            return null;
        }
        else {
            return Values.elementAt(index);
        }
    }

}

/*
 * Remove the record of the NSG with ID = key.
 *
 * @param key: Unique ID of NSG to be removed.
 */
public Object remove(Object key) {
    Object retVal;

    int index = Keys.indexOf(key);

    if(index == -1) {
        retVal = null;
    }
    else {
        Keys.removeElementAt(index);
        retVal = Values.elementAt(index);
        Values.removeElementAt(index);
    }

    return retVal;
}

/*
 * Provide an Enumeration data structure for the keys Vector.
 *
 */
public Enumeration keys() {
    return Keys.elements();
}

/*
 * Provide an Enumeration data structure for the elements Vector.
 *
 */
public Enumeration elements() {
    return Values.elements();
}

} // end of class NSGRepository

// end of file NSGRepository.java

```

APPENDIX E - XML FILE

```
<?xml version="1.0"?>

<!-- Simple XML file for use with the SOFT experiment -->
<!-- Note the hard-wired reference to the dtd. This should be fixed -->
<!-- in order to run on another system -->

<!DOCTYPE PROTOCOL_DESCRIPTION SYSTEM
"file:///DaB/Documentation/dabp/protocols/protocol_description.dtd">

<!-- SOFT/DABP XML PACKET DESCRIPTIONS -->

<PROTOCOL_DESCRIPTION>

<PROTOCOL_INFORMATION>
  <PROTOCOL_NAME>SOFT</PROTOCOL_NAME>
  <PROTOCOL_MARKER_FIELD>PacketType</PROTOCOL_MARKER_FIELD>
  <PROTOCOL_MARKER_POSITION>0</PROTOCOL_MARKER_POSITION>

  <PROTOCOL_MARKER_TYPE>org.web3d.vrtp.datatypes.UnsignedByte</PROTOCOL_MARKER_TYPE>

  <PROTOCOL_TYPE_LIST>
    <TYPE>org.web3d.vrtp.datatypes.UnsignedByte</TYPE>
    <TYPE>org.web3d.vrtp.datatypes.UnsignedShort</TYPE>
    <TYPE>org.web3d.vrtp.datatypes.SignedInteger</TYPE>
    <TYPE>org.web3d.vrtp.datatypes.DoublePrecision</TYPE>
    <TYPE>org.web3d.vrtp.datatypes.array6</TYPE>
    <TYPE>org.web3d.vrtp.datatypes.array16</TYPE>
    <TYPE>org.web3d.vrtp.datatypes.array40</TYPE>
  </PROTOCOL_TYPE_LIST>

  <!-- XXXX These fields are wrong! what should they be?! -->
  <PROTOCOL_HANDLER>http://www.stl.nps.navy.mil/~foo</PROTOCOL_HANDLER>

  <PROTOCOL_SEMANTICS_HANDLER>demo.dabp.Semantics</PROTOCOL_SEMANTICS_HANDLER>

</PROTOCOL_INFORMATION>

<ADUS>

<!-- XFORM PACKET -->
<ADU_DESCRIPTION>
<ADU_INFO>
  <ADU_NAME>XFORM</ADU_NAME>
  <ADU_MARKER_VALUE>1</ADU_MARKER_VALUE>
</ADU_INFO>

<FIELDS>

<FIELD_PRIMITIVE>
```



```

    <FIELD_NAME>PacketType</FIELD_NAME>
    <FIELD_TYPE>org.web3d.vrtp.datatypes.UnsignedByte</FIELD_TYPE>
    <FIELD_DEFAULT>1</FIELD_DEFAULT>
</FIELD_PRIMITIVE>

<FIELD_PRIMITIVE>
    <FIELD_NAME>Tag</FIELD_NAME>
    <FIELD_TYPE>org.web3d.vrtp.datatypes.array6</FIELD_TYPE>
    <FIELD_DEFAULT>9</FIELD_DEFAULT>
</FIELD_PRIMITIVE>

<FIELD_PRIMITIVE>
    <FIELD_NAME>CCode</FIELD_NAME>
    <FIELD_TYPE>org.web3d.vrtp.datatypes.SignedInteger</FIELD_TYPE>
    <FIELD_DEFAULT>0</FIELD_DEFAULT>
</FIELD_PRIMITIVE>

<FIELD_PRIMITIVE>
    <FIELD_NAME>NodeName</FIELD_NAME>
    <FIELD_TYPE>org.web3d.vrtp.datatypes.array40</FIELD_TYPE>
    <FIELD_DEFAULT>9</FIELD_DEFAULT>
</FIELD_PRIMITIVE>

<FIELD_PRIMITIVE>
    <FIELD_NAME>mat</FIELD_NAME>
    <FIELD_TYPE>org.web3d.vrtp.datatypes.array16</FIELD_TYPE>
    <FIELD_DEFAULT>9</FIELD_DEFAULT>
</FIELD_PRIMITIVE>

</FIELDS>
</ADU_DESCRIPTION>

</ADUS>
</PROTOCOL_DESCRIPTION>

```

APPENDIX F - DaBP CLIENTS

```
//-----
// Filename: SClient.java
// Date:      23-Aug-99
// Compiler:  JDK 1.2
//-----

import org.web3d.vrtp.dabp.*;
import org.web3d.vrtp.util.*;
import org.web3d.vrtp.net.*;

import java.net.*;
import java.io.*;
import java.util.*;

/**
 * SClient
 *
 * Simple client sending packets to the SOFT server.
 * It is used for the empirical testing.
 * SClient uses SOFTexp.xml in order to create the protocol description.
 * Then
 * it creates a byte array according to this protocol and sends this
 * 1000 times
 * to the server. The byte array contains the default data.
 * Syntax: SClient <server name> <server port>
 *
 * @author P.Fiabolis,G.Prokopakis
 */
public class SClient
{
    /**
     * Main method. Creates the protocol description, connects to the server
     * and
     * sends an ADU 1000 times to him.
     * We need to add a time delay between sending packets. Without this
     * delay
     * we get a NullPointerException exception (in the NSGServeOne) at the
     * ProtocolDescription.determineADUFromBinaryData method.
     */
    public static void main(String[] args) throws SocketException
    {
        // args[0] = Server name
        // args[1] = socket number

        if(args.length != 2) {
            usageMessage();
            System.exit(0);
        }

        ProtocolDescription protocol;
```

```

ADUData packet;

InetAddress addr;
Socket socket;
OutputStream os;
DataOutputStream dos;
ByteArrayOutputStream bos;

DataOutputStream me;

byte data[];

protocol = new ProtocolDescription("SOFTexp.xml");
packet = protocol.getADUDataForName("XFORM");

try
{
    addr = InetAddress.getByName(args[0]);
    System.out.println("addr = " + addr);
}
catch(UnknownHostException uhe)
{
    System.out.println("Unknown Host ...");
    return;
}

try
{
    socket = new Socket(addr, Integer.parseInt(args[1]));
}
catch(IOException ioe)
{
    System.out.println("Cannot create the socket ...");
    return;
}

try
{
    try
    {
        os = socket.getOutputStream();
    }
    catch(IOException e)
    {
        System.out.println("Could not initialize output stream
...");
        return;
    }
    PrintStream p = new PrintStream(os);
    bos = new ByteArrayOutputStream();
    dos = new DataOutputStream(bos);
    try {
me = new DataOutputStream(socket.getOutputStream());
    }
    catch(IOException ioe) {
        return;
    }
}

```

```

    }

    packet.serialize(dos);
    data = bos.toByteArray();

    for(int i = 0; i < 120000; i++)
    {
        try
        {
            me.write(data);
            me.flush();
        }
        catch(IOException ioe)
        {
            System.out.println("Could not write to the output stream
... " + i);
            return;
        }
    } // end while(true)

}
finally
{
    System.out.println("Closing client ...");
    try
    {
        socket.close();
    }
    catch(IOException e)
    {
        System.out.println("Could not close socket...");
        return;
    }
} // end try-finally
} // end main

/**
 * Provides a hint message when the number of the command line arguments
 * is wrong.
 */
public static void usageMessage() {
    System.out.print("Usage: java SClient <server name>");
    System.out.println(" <server port>");
    System.out.println();
    System.out.println("Exaple: java SClient venus 9000");
    System.out.println("\tvenus - machine running the SOFT server");
    System.out.println("\t9000 - venus machine port");
} // end usageMessage
} // end of class SClient

// end of file SClient.java

```

```

//-----
// Filename: RClient.java
// Date:      23-Aug-99
// Compiler:  JDK 1.2
//-----

import org.web3d.vrtp.dabp.*;
import org.web3d.vrtp.util.*;
import org.web3d.vrtp.net.*;

import java.net.*;
import java.io.*;
import java.util.*;

/**
 * RClient
 *
 * Simple client reading packets from the SOFT server.
 * It is used for the empirical testing.
 * RClient uses SOFTexp.xml in order to create the protocol description.
 * Then,
 * it reads ADUs from the server. No packet processing takes place.
 * Syntax: RClient <server name> <server port>
 *
 * @author P.Fiabolis,G.Prokopakis
 */
public class RClient
{
    /**
     * Main method. Creates the protocol description, connects to the server
     * and
     * reads ADUs from him.
     */
    public static void main(String[] args)
    {
        // args[0] = Server name
        // args[1] = socket number

        if(args.length != 2) {
            usageMessage();
            System.exit(0);
        }

        ProtocolDescription protocol;
        ADUData packet;

        InetAddress addr;
        Socket socket;
        OutputStream os;
        DataOutputStream dos;
        ByteArrayOutputStream bos;

        ADUStreamTCP tcpStream;
        ADUData data;
    }
}

```

```

protocol = new ProtocolDescription("SOFTexp.xml");
packet = protocol.getADUDataForName("XFORM");

try
{
    addr = InetAddress.getByName(args[0]);
    System.out.println("addr = " + addr);
}
catch(UnknownHostException uhe)
{
    System.out.println("Unknown Host ...");
    return;
}

try
{
    socket = new Socket(addr, Integer.parseInt(args[1]));
}
catch(IOException ioe)
{
    System.out.println("Unknown Host ...");
    return;
}

tcpStream = new ADUStreamTCP(socket, protocol);

try
{
    while(true)
    {
        data = tcpStream.readNextADU(67);
        if(data != null) {
            System.out.print(".");
        }
    } // end while(true)
}
finally
{
    System.out.println("Closing client ...");
    try
    {
        socket.close();
    }
    catch(IOException e)
    {
        System.out.println("Could not close socket...");
        return;
    }
} // end try-finally

} // end main

/**
 * Provides a hint message when the number of the command line arguments
 * is wrong.

```

```

*
*/
public static void usageMessage() {
    System.out.print("Usage: java RClient <server name>");
    System.out.println(" <server port>");
    System.out.println();
    System.out.println("Exaple: java RClient venus 9000");
    System.out.println("\tvenus - machine running the SOFT server");
    System.out.println("\t9000 - venus machine port");
}

} // end of class RClient

// end of file RClient.java

```

APPENDIX G - JAVA CODE FOR "PHASE FIVE"

```
//-----  
// Filename: NSGServer.java  
// Date:      28-Aug-99  
// Compiler:  JDK 1.2  
//-----  
  
import org.web3d.vrtp.dabp.*;  
import org.web3d.vrtp.util.*;  
  
import java.net.*;  
  
import java.io.*;  
  
/**  
 * Main class for the NSG server. Enters an endless loop waiting  
 * for client requests.  
 *  
 * @author P.Fiabolis,G.Prokopakis  
 */  
public class NSGServer {  
  
    /*  
     * App main function.  
     *  
     * @param args: Command line param. Should have the port number.  
     * @exception IOException  
     */  
    public static void main (String[] args) throws IOException {  
  
        ProtocolDescription protocol;  
  
        if(args.length != 1) {  
            usageMessage();  
            System.exit(0);  
        }  
  
        protocol = new ProtocolDescription("SOFTexp.xml");  
        ServerSocket server_socket;  
        InetAddress addr;  
  
        int counter = 1;  
  
        try {  
            server_socket = new ServerSocket(Integer.parseInt(args[0]));  
  
            while(true) {  
                Socket in_socket = server_socket.accept();  
  
                addr = in_socket.getInetAddress();  
                System.out.println("Welcome " + addr.getHostName());  
            }  
        }  
    }  
}
```



```

        try {
            new NSGServeOne(in_socket, counter++, protocol);
        }
        catch(IOException e) {
            in_socket.close();
        }
    } // end while
}
catch(IOException e) {
    System.out.println("Cannot initialize server. Exiting ...");
    System.exit(0);
}

} // end of main

/*
 * Message to display in case of invalid command line arguments.
 */
public static void usageMessage() {
    System.out.println("Usage : java Server <port number>");
    System.out.println();
    System.out.println("Example: java Server 9999");
} // end of usageMessage

} // end of class NSGServer

// end of file NSGServer.java

//-----
// Filename: NSGServeOne.java
// Date: 20-Aug-99
// Compiler: JDK 1.2
//-----

import org.web3d.vrtp.dabp.*;
import org.web3d.vrtp.util.*;

import java.io.*;
import java.net.*;
import java.util.Hashtable;
import java.util.Enumeration;
import java.util.Vector;

import java.util.Calendar;

```

```

/**
 * NSGServeOne
 *
 * An instance of this class is running for every client connected to
 the
 * NSGServer. Reads from an ADUStreamTCP stream. Each ADU is transformed
 to
 * a byte array and retransmitted to the other clients currently
 connected.
 *
 * @author P.Fiabolis,G.Prokopakis
 */

class NSGServeOne extends Thread {

/**
 * Unique ID for each thread. For the moment this is a simple counter
 */
int threadID;

/**
 * Stream where this thread will write the NSG's stored in the
 repository.
 */
OutputStream me;

/**
 * Table holding the ports for all conected clients. Common for all
 threads
 */
static Vector castTable = new Vector();

/**
 * Table holding SG data sent from clients. Common for all threads
 */
static NSGRepository table = new NSGRepository();

/**
 * Semaphore for thread exit.
 */
boolean stopThread = false;

/**
 * Socket associated with this thread.
 */
Socket threadSocket;

/**
 * DaBP stream to read from. We can read ADUs directly from here.
 */
ADUStreamTCP tcpStream;

/**
 * Represents one ADU read from the network.
 */
ADUData data;

```

```

/*
 * NSGServeOne
 *
 * Class constructor. Initialize input and output streams, adds this
client
 * to the client pool, and starts the thread execution.
 * A NullPointerException occurs after reading a number of packets
(265).
 * Reason is not known for the moment but it is eliminated when a
 * time delay is added to the sending client (SClient) between packet
 * sending.
 *
 * @param in_socket: Socket for this thread connection
 * @param id: unique ID for this client. For the moment just a counter
 * @exception IOException
 */
public NSGServeOne(Socket in_socket, int id, ProtocolDescription
protocol)
    throws IOException {

    threadID = id;
    threadSocket = in_socket;

    /* // try to increase the input buffer size
        System.out.println("Buffer size 1 = " +
threadSocket.getReceiveBufferSize());

threadSocket.setReceiveBufferSize(threadSocket.getReceiveBufferSize() *
3);
        System.out.println("Buffer size 2 = " +
threadSocket.getReceiveBufferSize());
    */
    me = new DataOutputStream(in_socket.getOutputStream());

    // Add the new comer to the client list.
    castTable.addElement(me);

    tcpStream = new ADUStreamTCP(in_socket, protocol);

    start();
}

/*
 * Waits until a packet arrives. Then, it starts reading ADUs, stores
them, and
 * retransmits them to the other clients. When done (1000 reads)
calculates
 * the duration.
 *
 */
public void run() {

    int len = 0;
    OutputStream out;

```

```

Calendar begin, end;

byte[] byteData;

try {

    // late-comer support not needed for this experiment.

    InputStream is;

    // Wait until there is something available to read.
    while(len == 0)
    {
        try
        {
            is = threadSocket.getInputStream();
            len = is.available();
        }
        catch(IOException ioe)
        {
            System.out.println("Cannot get input stream ...");
            closeThread();
        }
    } // end while(len == 0)

    // Start the timer.
    begin = Calendar.getInstance();
    System.out.println("Timer started ...");

    int counter = 0;

    while(!stopThread)
    {
        try
        {
            data = tcpStream.readNextADU(67);

            if(data != null) {

                try {
                    Object val = table.put(data.get("Tag"), data);
                }
                catch(FieldNotFoundException fnf) {
                    System.out.println("Field not found ...");
                }

                Enumeration enum = castTable.elements();

                byteData = data.getBinaryData();

                while(enum.hasMoreElements()) {
                    out = (OutputStream)enum.nextElement();
                    if (out != me) {
                        out.write(byteData);
                    }
                }
            }
        }
    }
}

```

```

        } // end while

        counter++;
        if(counter >= 120000) {
            end = Calendar.getInstance();

            long startTime =
(begin.get(Calendar.HOUR)*60*60*1000) +
(begin.get(Calendar.MINUTE)*60*1000) +
(begin.get(Calendar.SECOND)*1000) +
(begin.get(Calendar.MILLISECOND));
            long endTime =
(end.get(Calendar.HOUR)*60*60*1000) +
(end.get(Calendar.MINUTE)*60*1000) +
(end.get(Calendar.SECOND)*1000) +
(end.get(Calendar.MILLISECOND));

            long duration = endTime - startTime;

            System.out.println("Duration = " + duration );

            System.out.println("Goodbye " + threadID);
            System.out.println("1000 packets read ...");
            closeThread();
        }
    } // end if
}
catch(IOException eee)
{
    System.out.println("Goodbye " + threadID);
    System.out.println("Could not read next ...");
    closeThread();
}

} // end while(true)
}
finally {
    closeThread();
}
}

/*
 * Close the thread after finishing connection or failure.
 */
public void closeThread() {
    try {
        threadSocket.close();
        castTable.removeElement(me);
    }
}

```

```
        stopThread = true;
    }
    catch(IOException e) {
        System.out.println("ERROR: Could not close socket @
closeThread");
    }
}

} // end NSGServeOne
// end of file NSGServeOne.java
```


APPENDIX H - JAVA CODE FOR TESTING CLIENTS

```
//-----  
// Filename: SClient.java  
// Date: 24-August-99  
// Compiler: JDK 1.2  
// Comments: Simple client sending packets to the SOFT server  
// Used for the empirical testing.  
//-----  
  
import java.net.*;  
import java.io.*;  
import java.util.*;  
  
public class SClient  
{  
    public static void main(String[] args)  
    {  
        // args[0] = Server name  
        // args[1] = socket number  
  
        if(args.length != 2) {  
            usageMessage();  
            System.exit(0);  
        }  
  
        InetAddress addr;  
        Socket socket;  
        FileInputStream fis;  
        DataOutputStream dos;  
  
        try  
        {  
            addr = InetAddress.getByName(args[0]);  
            System.out.println("addr = " + addr);  
        }  
        catch(UnknownHostException uhe)  
        {  
            System.out.println("Unknown Host ...");  
            return;  
        }  
  
        try  
        {  
            socket = new Socket(addr, Integer.parseInt(args[1]));  
        }  
        catch(IOException ioe)  
        {  
            System.out.println("Cannot create the socket ...");  
            return;  
        }  
  
        try  
        {  

```



```

client.    // packet.exp contains a byte array extracted from a C++
           // this packet corresponds to a transformation.
           fis = new FileInputStream("packet.exp");
       }
       catch(FileNotFoundException fnfe)
       {
           System.out.println("File packet.exp not found ...");
           try
           {
               socket.close();
           }
           catch(IOException ioe)
           {
               System.out.println("Can not close socket ...");
           }
           return;
       }

       byte[] b = new byte[65535];
       int len;

       try
       {
           len = fis.read(b);
           dos = new DataOutputStream(socket.getOutputStream());

           for(int i = 0; i < 120000; i++)
           {
               dos.writeInt(len);
               dos.write(b, 0, len);
           } // end for

           System.out.println("Done");
       }
       catch(IOException ioe)
       {
           System.out.println("Can not read file ...");
       }

       try
       {
           socket.close();
       }
       catch(IOException ioe)
       {
           System.out.println("Can not close socket ...");
       }
       return;
   }

   } // end main

public static void usageMessage() {
    System.out.print("Usage: java SClient <server name>");
}

```

```

        System.out.println(" <server port>");
        System.out.println();
        System.out.println("Exaple: java SClient venus 9000");
        System.out.println("\tvenus - machine running the SOFT server");
        System.out.println("\t9000 - venus machine port");
    }

} // end of class SClient

```

```

//-----
// Filename: RClient.java
// Date:      24-August-99
// Compiler:  JDK 1.2
// Comments:  Simple client receiving packets from the SOFT server
//           Used for the empirical testing.
//-----

import java.net.*;
import java.io.*;

public class RClient
{
    public static void main(String[] args) throws IOException
    {
        // args[0] = Server name
        // args[1] = socket number

        if(args.length != 2) {
            usageMessage();
            System.exit(0);
        }

        InetAddress addr;
        Socket socket;
        DataInputStream in;

        try
        {
            addr = InetAddress.getByName(args[0]);
            System.out.println("addr = " + addr);
        }
        catch(UnknownHostException uhe)
        {
            System.out.println("Unknown Host ...");
            return;
        }

        try
        {
            socket = new Socket(addr, Integer.parseInt(args[1]));
        }
    }
}

```

```

        catch(IOException ioe)
        {
            System.out.println("Cannot create the socket ...");
            return;
        }

        byte[] b = new byte[65535];
        int len;

        in = new DataInputStream (socket.getInputStream());

        try
        {
            while(true)
            {
                len = in.readInt();

                byte[] buf = new byte[len];

                in.readFully(buf, 0, len);

            } // end while(true)
        }
        finally
        {
            try
            {
                socket.close();
            }
            catch(IOException ioe)
            {
                System.out.println("Can not close socket ...");
                return;
            }
        }

    } // end main

    public static void usageMessage() {
        System.out.print("Usage: java RClient <server name>");
        System.out.println(" <server port>");
        System.out.println();
        System.out.println("Exaple: java RClient venus 9000");
        System.out.println("\tvenus - machine running the SOFT server");
        System.out.println("\t9000 - venus machine port");
    }

} // end RClient

```

LIST OF REFERENCES

- [ALEX78] Alexander, A. 1978. "Impacts of Telemation on Modern Society." Proceedings 1st IFToMM Symposium, Vol. 2.
- [BROL97] Broll, Wolfgang. 1997. "Populating the Internet: Supporting Multiple Users and Shared Applications with VRML." Proceedings of the VRML'97 Symposium (Monterey, CA, 24-26 February, 1997).
- [CAPP96] Capps, M. et al. 1996. "Distributed Interoperable Virtual Environments." Proceedings of the Third International Conference on Configurable Distributed Systems (Annapolis, Maryland, 6-8 May, 1996).
- [CHEW98] Chew, F. 1997. The Java/C++ Cross-Referencer Handbook. Prentice Hall.
- [DeFA98] DeFanti, T. et al. 1998. Personal Tele-Immersion Devices. IEEE.
- [DERT98] Dertouzos, M. 1998. What Will Be. Harper Collins, New York, NY.
- [ECKE98] Eckel, B. 1998. Thinking in Java. Prentice Hall.

- [EDWA97] Edwards, J. and DeVoe, Deborah. 1997. 3-tier Client/Server at work. John Wiley & Sons.
- [FLAN97] Flanagan, D. 1997. Java in a Nutshell. O'Reilly and Associates.
- [FOST98] Foster, I. and Kesselman, C. 1998. The Grid: Blueprint for a New Computing Infrastructure. Morgan Kaufmann Publishers.
- [FUCH98] Fuchs, H. et al. 1998. "The Office of the Future: A Unified Approach to Image-Based Modeling and Spatially Immersive Displays." Proceedings SIGGRAPH 98 Conference. Annual Conference Series. ACM SIGGRAPH, Orlando, FL.
- [HILL97] Hill, J. and Jensen, J. 1998. "Telepresence Technology in Medicine: Principles and Applications." Proceeding of the IEEE, Vol. 86, No. 3, March 1998.
- [LEA96] Lea, R. et al. 1996. Java for 3D and VRML Worlds. New Riders Publishing.
- [LEA97] Lea, R. et al. "Community Place: Architecture and Performance." Proceedings of the VRML'97 Symposium (Monterey, CA, 24-26 February, 1997).
- [MACI98] MacIntyre, B. and Feiner S. 1998. "A Distributed 3D Graphics Library." Computer

Graphics Proceedings, Annual Conference Series of 1998.

- [MYER97] Meyers, S. 1996. More Effective C++. Addison Wesley.
- [PAUL98] Paul, S. 1998. Multicast on the Internet and Its applications. Kluwer Academic Publishing.
- [PIER98] Pierce, J. et al. 1998. "Image Plane Interaction Techniques in 3D Immersive Environments." Proceedings of 1997 Symposium on Interactive 3D Graphics, (Providence, Rhode Island, April 27-30, 1997).
- [SCHA99] Schach, S. 1999. Classical and Object-Oriented Software Engineering with UML and Java. McGraw-Hill, Fourth Edition.
- [STAL98] Stallings, W. 1998. Operating Systems, Internals and Design Principles. Prentice Hall, Third Edition.
- [TOFF80] Toffler, A. 1980. The Third Wave. Bantam Books.
- [ZYDA97] Zyda, M. and Macedonia, M. 1997. "A Taxonomy for Networked Virtual Environments." Periodical IEEE Multimedia, Vol. 4, No. 1, January - March 1997.

[WEB]

- 1 Www.cis.upenn.edu/~kamberov/doc/teleimmersion.html
- 2 Www.internet2.edu/html/mission.html
- 3 Www.sgi.com/fahrenheit/scene.pdf
- 4 Www.pyramidsystems.com
- 5 Www.advanced.org/teleimmersion/board/cubelabel.html
- 6 Www.cs.unc.edu/Research/stc/teleimmersion/index.html
- 7 Www.advanced.org/teleimmersion/board/cubelover.html
- 8 Www.evl.uic.edu/pape/CAVE
- 9 Www.pyramidsystems.com/idesk.html
- 10 Www.iuinfo.indiana.edu/ocm/releases/coxuits.htm
- 11 Web.mit.edu/hmsl/www/Telesurgery/hardware.html
- 12 Www-dse.doc.ic.ac.uk/~nd/surprise_96/journal/vol4/ao2/report.html#safe_surgery
- 13 Web.mit.edu/hmsl/www/markott/cooptelesurg.html
- 14 Cs.franklin.edu/Faculty/Giuliani/mba682/~cgriggs/tsld003.htm
- 15 Www.merl.com/projects/spline
- 16 Www.java.sun.com/products/jdk/1.2/docs/api

BIBLIOGRAPHY

Berzins, V. 1995. *Software Merging and Slicing*. IEEE.

Booch, G. and Rumbaugh, J. and Jacobson, I. 1999. *The Unified Modeling Language User Guide*. Addison Wesley.

Brooks, F. 1995. *The Mythical Man-Month, Second Edition*. Addison-Wesley.

Couch, J. 1999. *JAVA 2 Networking*. McGraw-Hill.

D'Souza, D. and Wills A. 1999. *Objects, Components and Frameworks with UML*. Addison Wesley.

Deitel, H. and Deitel, P. 1994. *C++ How to Program*. Prentice Hall.

Deitel, H. and Deitel, P. 1998. *JAVA How to Program*. Prentice Hall.

Eckel, B. 1998. *Thinking in JAVA*. Prentice Hall.

Edwards, J. 1997. *3-Tier Client/Server At Work*. John Wiley.

Flanagan, D. 1997. *JAVA in a Nutshell, Second Edition*.
O'Reilly.

Foster, I. And Kesselman, C. 1999. *The Grid: Blueprint for a
New Computing Infrastructure*. Morgan Kaufmann.

Gordon, R. 1998. *Essential JNI, Java Native Interface*.
Prentice Hall.

IEEE. 1997. *Software Engineering*. IEEE Standards Collection.

Larman, G. 1998. *Applying UML and Patterns*. Prentice Hall.

Lea, R. and Matsuda, K. and Miyashita, K. 1996. *Java for 3D
and VRML Worlds*. New Riders.

Meyers, S. 1996. *More Effective C++*. Addison Wesley.

Meyers, S. 1998. *Effective C++, Second Edition*. Addison
Wesley.

Myers, G. 1979. *The Art of Software Testing*. Wiley.

Oaks, S. and Wong, H. 1999. *JAVA Threads, Second Edition*. O'Reilly.

Paul, S. 1998. *Multicasting on the Internet and Its Applications*. Kluwer Academic Publishers.

Rago, S. 1993. *UNIX System V Network Programming*. Addison Wesley.

Robbins, K. and Robbins, S. 1996. *Practical UNIX Programming, A Guide to Concurrency, Communication and Multithreading*. Prentice Hall.

Rumbaugh, J. and Jacobson, I. and Booch, G. 1999. *The Unified Modeling Language Reference Manual*. Addison Wesley.

Schach, S. 1999. *Classical and Object-Oriented Software Engineering with UML and C++, Fourth Edition*. McGraw-Hill.

Schach, S. 1999. *Classical and Object-Oriented Software Engineering with UML and JAVA, Fourth Edition*. McGraw-Hill.

Stallings, W. 1998. *Operating Systems, Internals and Design Principles*. Prentice Hall, Third Edition.

Stevens, R. 1993. *Advanced Programming in the UNIX Environment*. Addison Wesley.

Stevens, R. 1998. *UNIX Network Programming, Networking API's: Sockets and XTI, Vol 1, Second Edition*. Prentice Hall.

Stevens, R. 1999. *UNIX Network Programming, Interprocess Communication, Vol 2, Second Edition*. Prentice Hall.

Stroustrup, B. 1997. *The C++ Programming Language, Third Edition*. Addison Wesley.

INITIAL DISTRIBUTION LIST

- 1 Defense Technical Information Center 2
8725 John J. Kingman Rd., STE 0944
Ft. Belvoir, VA 22060-6218
- 2 Dudley Knox Library 2
Naval Postgraduate School
411 Dyer Rd.
Monterey, CA, 93943-5101
- 3 Professor, Michael J. Zyda. 1
Code CS/Zk
Naval Postgraduate School
Monterey, CA, 93943
- 4 Research Assistant, Michael V. Capps 1
Code CS/Ca
Naval Postgraduate School
Monterey, CA, 93943
- 5 Senior Lecturer, John S. Falby. 1
Code CS/Fa
Naval Postgraduate School
Monterey, CA, 93943
- 6 Jaron Lanier. 1
Advanced Network Services
200 Business Park Drive
Armonk, NY 10504
- 7 Brigadier Constantinos Gerokostopoulos 1
Research and Informatics Corps
Hellenic Army General Staff
Stratopedo Papagou
Holargos 15561
GREECE
- 8 Panagiotis Fiambolis 1
Ithakis & Dodekanisou 2A
Gerakas 15344
GREECE

- 8 George Prokopakis 1
Smyrnis 1
Zografou 15772
GREECE
- 9 Chairman, Code CS 1
Naval Postgraduate School
Monterey, CA, 93943